VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Podobnost softwarových artefaktů

Software Artifact Similarity

2013

Bc. Ondřej Dočkal

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Ondřej Dočkal**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Podobnost softwarových artefaktů
Software Artifact Similarity

Zásady pro vypracování:

Softwarové artefakty (programy, knihovny) vznikají za cenu značných nákladů. Neoprávněné kopírování autorsky chráněných zdrojových kódů proto je velmi nežádoucí činnost.

Cílem práce je návrh a implementace algoritmu pro vyhodnocení podobnosti softwarových artefaktů. Podobnost bude vyhodnocena na několika úrovních: podobnost zdrojových kódů, podobnost konfigurace (použité knihovny atd.), podobnost mezikódu (např. Java bytekód, CIL) a podobnost dekompilovaných kódů.

Seznam doporučené odborné literatury:

[1] Source Code Similarity Detection Using Adaptive Local Alignment of Keywords by Jeong-Hoon Ji, Soo-Hyun Park, Gyun Woo, Hwan-Gue Cho, Artificial Intelligence (2007) ,IEEE Computer Society, Pages: 179-180, ISBN: 0769530494, DOI: 10.1109/PDCAT.2007.75

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Pavel Krömer, Ph.D.**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2013 .....

V první řadě bych rád poděkoval svému vedoucímu této práce Ing. Pavlu Krömerovi, Ph.D. za jeho čas, hodnotné rady a připomínky, které mi velice pomohly při vypracovávání této diplomové práce. Dále patří poděkování „plagiátorům“, kteří vytvořili část sady testovacích dat, a také přátelům a kolegům za motivování při psaní. Práce je věnována rodině, přátelům, místním i zahraničním, a nejbližším, kteří mě podporovali, inspirovali, a nevědomky tak pomohli dokončit následující řádky.

Abstrakt

Tato diplomová práce se zabývá problematikou plagiátorství ve zdrojových kódech. Teoretická část práce popisuje plagiátorství z obecného hlediska, uvádí jeho formy, typy a příklady. Součástí teoretického rozboru, který se nachází v třetí kapitole, je popis detekčních přístupů a algoritmů, které vyhodnocují podobnost softwarových artefaktů. Kapitola čtvrtá se věnuje algoritmu adaptivního lokálního zarovnání klíčových slov. Na základě analýzy teoretické části navrhuji detekční proces, který využívá adaptivní lokální zarovnání. Návrh a implementace aplikace, která vyhodnocuje podobnost programů na úrovních zdrojových kódů, bytekódů, konfigurace a dekompilovaných kódů, je popsána v páté a šesté části.Testování aplikace pro detekci plagiátů na několika úrovních je vyhodnoceno v sedmé kapitole. Závěrečná kapitola sumarizuje dosažené cíle práce.

Klíčová slova: plagiátorství, detekce plagiátů, softwarový artefakt, adaptivní lokální zarovnání, Smith - Watermanův algoritmus, detekční techniky, vyhledávání podobnosti, bytekód, dekompilovaný kód, tokenizace

Abstract

This thesis deals with the plagiarism of source code. The theoretical part of the thesis describes plagiarism in general, provides its forms, types and examples. The part of theoretical analysis, which is located in the third chapter, is description of detection approaches and algorithms that evaluate the similarity of software artifacts. The fourth chapter describes the algorithm of adaptive local alignment of the keywords. I propose, based on the theoretical analysis, a detection process that uses adaptive local alignment. Design and implementation of the application which evaluates the similarity of the programs on different levels, such as source code, bytecode, configuration and decompiled source codes, is described in the fifth and sixth section. Evaluation of the plagiarism detection, on several levels within applications, is evaluated in the seventh section. The final chapter summarizes achieved goals, defined in the beginning.

Keywords: plagiarism, plagiarism detection, software artefact, adaptive local alignment, Smith - Waterman algorithm, detection techniques, similarity evalutation, bytecode, decompiled code, tokenization

Seznam použitých zkratek a symbolů

JFlex	–	The Fast Scanner Generator for Java
ČSN	–	Česká Soustava Norem
ISO	–	International Standard Organisation
CUP	–	Create Useful Parser
PDS	–	Plagiarism Detection System
HTML	–	Hyper-Text Markup Language
GNU	–	GNU's Not Unix
ALA	–	Adaptive Local Alignment
AST	–	Abstract Syntax Tree
PDG	–	Program Dependence Graph
SCO	–	The Santa Cruz Operation
PDA	–	Plagiarism Detection Application
SVR4	–	System V version 4
SQL	–	Structured Query Language
YACC	–	Yet Another Compiler Compiler
ANTLR	–	Another Tool for Language Recognition
GST	–	Greedy String Tiling
YAP	–	Yet Another Plague
MOSS	–	Measure Of Software Similarity
GUI	–	Graphical User Interface
JVM	–	Java Virtual Machine
XML	–	Extensible Markup Language
SW	–	Smith - Waterman algorithm
UML	–	Unified Modeling Language
JAD	–	Java Decompiler
API	–	Application Programming Interface

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Plagiátorství je krádež duševního vlastnictví původního autora. Člověk, který takto použije myšlenky někoho jiného se nedopouští jenom zcizení, ale navíc i přivlastnění jejich autorství. Tohoto přečinu se lidé dopouští z různých důvodů a již hodně dlouhou dobu. Na počátku bylo slovo, pak psané slovo, následovalo umění, knihy, vědecké práce, multimédia a další artefakty. V této práci se budu zabývat těmi softwarovými artefakty a způsobem, jak odhalit jejich neoprávněné užití.

Problematika plagiátorství ve zdrojových kódech je v porovnání s ostatními odvětvími velice mladá. Prozkoumání všech možností při odhalování plagiátů, které nám nabízí obor informačních technologií, je výzvou už jenom díky vynalézavosti těch, co se snaží obejít morální zásady a různé ochrany, protože k jejich odhalování existuje nepřeberné množství technik a postupů. Tato práce se zabývá návrhem a implementací algoritmu pro vyhledání plagiátů ve zdrojovém kódu.

1.1 Motivace

Počítačové programy, aplikace, knihovny či frameworky, všechny tyto softwarové artefakty vznikají s velkým úsilím programátorů, softwarových inženýrů, testerů, návrhářů a dalších lidí zapojených do vývoje softwarového produktu. Každý takto vytvořený produkt je chráněn autorským zákonem. Nelze tedy přehlížet neoprávněné kopírování, zneužití těchto prací nebo jejich částí.

1.2 Analýza problematiky

Většina dnešních přístupů a nástrojů porovnává vstupní programy na jedné úrovni. Jedná se většinou o původní zdrojové kódy. V literatuře se nachází široké množství technik, které se zabývají vyhledáváním plagiátů na úrovni strojového kódu nebo interakce programu se systémem. Při studování problematika jsem nenarazil na takový nástroj, který integruje několik takových úrovní najednou.

Detekce podobnosti bude probíhat mezi dvěma programy, které jsou napsány v programovacím jazyce Java. Systém zpracuje vstupní soubory do vhodné vnitřní struktury tak, aby použitý srovnávací algoritmus vyhodnotil možné plagiátorské modifikace s nejvyšší přesností a mírou podobnosti. Vyhodnocení podobnosti bude probíhat na několika úrovních, z pohledu Javy se jedná o zdrojový kód, strojový kód neboli bytekód, dekompilovaný zdrojový kód a konfigurace programu. Aplikace bude vizualizovat procentuální podobnost a vypíše podezřelé fragmenty kódu.

1.3 Návrh řešení

Systém bude provádět sekvenci procesů, které vyhodnotí podobnost dvou programů. Na začátku se postupně načtou vstupní data do paměti, vytřídí irelevantní oblasti, které

nejsou důležité při detekci plagiátů, provedou se základní formátovací nebo normalizační postupy, které odstraní další nepodstatné elementy z analyzovaných dat a následně budou transformovány do vnitřní struktury, vhodné pro detekční algoritmus. Ten vyhodnotí míru podobnosti různých fragmentů a označí části původního kódu, který budeme chtít vizualizovat. V závěru aplikace vygeneruje přehledný, strukturovaný výpis, na jehož základě posoudí zodpovědná osoba, zdali byly označené fragmenty plagiarizovány.

1.4 Přínos a struktura práce

Ve své práci integruji nové a vyzkoušené přístupy detekce plagiátů, zabývám se víceúrovňovým srovnáním programů, kdy nahlížím na samotný zdrojový kód v několika jeho fázích překladu a využívám tento vývoj z pohledu vyhledávání plagiátů. Porovnáním výsledků z těchto úrovní a mezi nimi, zkoumám nové možnosti v oblasti detekce plagiátů.

Práce je logicky rozdělena do dvou hlavních částí, teoretické a praktické. Kapitola první Vás provede úvodem do této práce. V druhé kapitole se zabývám problematikou plagiátorství, je řečeno, co to jsou plagiáty, kde se mohou vyskytnout, jaká je jejich definice a jak jsou ošetřeny naším legislativním rámcem. Nastiňuji dvě hlavní odvětví plagiátorství (textové a softwarové) a uvádím reálný případ ze světa softwaru. Třetí kapitola popisuje způsoby, kterýma plagiátor modifikuje zdrojový kód a jakým způsobem pracují procesy, které tyto změny odhalují. Blíže se budu věnovat algoritmům, které se využívají v detekci plagiátů. Nakonec popíšu některé známé detekční nástroje. Kapitola č. 4 je na pomezí mezi teoretickou a praktickou částí, neboť popisuje další algoritmy pro vyhledávání plagiátů, které jsem zároveň použil ve vlastní aplikaci. Pátá kapitola obsahuje podrobnější požadavky aplikace detekce plagiátů a návrh její implementace. V šesté kapitole jsou pospány jednotlivé procesy aplikace z pohledu její implementace v jazyce Java. Poslední část práce popisuje výsledky reálného testování mého vlastního systému a porovnává je s JPlag aplikací a v jejím závěru hodnotím dosažené výslekdy a navrhuji možná rozšíření a vylepšení. Závěr všechny předchozí části hodnotí ve srovnání se zadáním.

2 Taxonomie plagiátorství

Co je to vlastně plagiátorství? Jak jej můžeme spolehlivě definovat? Čeho se může týkat a kde se vyskytuje? Jaké jsou hranice, kdy lze říci, že se jedná o plagiát a kdy nikoliv. V této části práce se zaměřuji na definování a druhy plagiátů, oblasti, kde můžeme o plagiátorství hovořit, a také nahlédnu na právní stránku této problematiky, jak je dána českým právním řádem.

Plagiátorství se ovšem nevyskytuje pouze v dílech psaných v přirozeném jazyce, jak by se dalo předpokládat, ale také v programovacích jazycích a týká se softwarových artefaktů, které jsou přístupné pod rozličnými licencemi, a to převážně v oblasti open source prostředí. V tomto případě se budu zabývat soudním sporem o práva k UNIXu a také si popíšu plagiátorství na akademické půdě.

2.1 Plagiátorství

Pokud někdo reprodukuje část nebo celou práci, případně dílo někoho jiného, aniž by správně uvedl zdroj, ze kterého pochází, dopouští se plagiátorství [15]. Na akademické půdě se tento jev objevuje nejvíce tam, kde studenti při vypracovávání svých prací, projektů a úkolů používají nejrůznější zdroje bez řádného uvedení v referencích. Čerpat přitom mohou z nejrůznějších informačních bází, ať už z knihoven, vědeckých žurnálů anebo z Internetu, kde se dnes nachází v podstatě všechny zdroje výše zmíněné. Většinou se studenti dopouští plagiátorství nevědomě, když špatně citují zdroje. Méně časté je záměrné kopírování cizí práce. Tyto případy se mohou týkat jak textových dokumentů, tak i zdrojových kódů a jiných softwarových artefaktů.

Plagiátorství definuje i mezinárodní norma ČSN ISO 5127-2003:

Definice 2.1 *Jedná se o „představení duševního díla jiného autora půjčeného nebo napodobeného v celku nebo z části, jako svého vlastního“ [27].*

Jinými slovy „užití jakékoliv myšlenky v odborné práci, bez uvedení jejího autora je plagiátorství“. Čímž se dopouští porušení etiky a autorského zákona [13].

2.2 Plagiát

Plagiátem je podle České terminologické databáze: „Nedovolená napodobenina (přesná nebo částečná) uměleckého nebo vědeckého díla jiné osoby, která je bez uvedení předlohy vydávána za originál; její původce tak porušuje autorská práva původní předlohy“ [28].

Ačkoli tato terminologie dostatečně označuje, co je plagiát a co není, měli bychom si ujasnit, čeho se může týkat. Pojem originál totiž může zahrnovat jak vědecké, tak umělecké dílo: textový dokument, zdrojový kód, návrhový design aplikace, umělecké dílo, ať už obraz nebo sochu, fotografii, audiovizuální nahrávku, atd.

Další typ práce, která sice není plagiátem, ale zvláště na akademické půdě je blízko této hranici, je kompilace. Jedná se o odborný dokument, který vznikl na základě poznatků různých prací, ale vše řádně cituje a nevydává tyto myšlenky za vlastní. Podle [5] zde spočívá prostor pro plagiarizování, převážně ve schopnosti studentů rozlišit hranici mezi plagiátem a kompilací.

2.3 Legislativní rámec plagiátorství

V autorském zákoně se výrazy plagiát a plagiátorství nenachází. Nicméně plagiátorství je přímým porušením tohoto zákona [5]. V České republice je závaznou legislativní normou a zároveň ochranou autorského práva zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů [13], který mimo jiné říká: „Předmětem práva autorského je dílo literární a jiné dílo umělecké a dílo vědecké, které je jedinečným výsledkem tvůrčí činnosti autora a je vyjádřeno v jakékoli objektivně vnímatelné podobě včetně podoby elektronické, trvale nebo dočasně, bez ohledu na jeho rozsah, účel nebo význam.“[5]

Plagiátorství na vysokých školách je přestupkem disciplinárním a upravují jej zákon o vysokých školách (zákon č. 111/1998 Sb.) a vnitřní směrnice školy. Postihy za porušení akademické etiky se mohou různit, od upuštění od postihu (například pokud se jedná o malý přestupek z nedbalosti), napomenutí až po vyloučení ze studií [16]. Pokud je plagiátorství zjištěno dodatečně po udělení titulu, v geografii České republiky není možné nabytý VŠ titul odebrat.

Plagiátorství je postižitelné také jako porušení trestního zákona (zákon č. 140/1961 Sb.), kdy se lze odvolat na paragrafy o porušení autorského práva, poškozování cizích práv a podvod.

2.4 Rozdělení plagiátorství

Plagiátorství můžeme rozdělit podle typu a formy. V první kategorii se plagiátorství rozděluje na úmyslné a neúmyslné [29].

2.4.1 Rozdělení plagiátorství podle typu

Plagiátorství úmyslné

Úmyslným plagiátorstvím se již podle názvu rozumí vědomé porušení intelektuálního vlastnictví původního autora. Plagiátor tak činí za účelem usnadnění vypracování vlastní intelektuální práce. Takové jednání je nečestné, navíc porušuje etiku a řád (vzdělávací instituce nebo společnosti) a hlavně autorské právo [5]. Následuje definice úmyslného plagiátorství podle Mareše [29].

Definice 2.2 *Jedná se o „doslovné opsání nebo počítačové zkopírování celého textu či hlavních pasáží textu jiného autora (nebo jiných autorů) se snahou vydávat je za svůj vlastní. Aktér neuvádí ani autora originálního textu, ani necituje původní pramen.“*

Formy úmyslného plagiátorství

V této kategorii se může vyskytovat několik forem, jakými se plagiátor dopouští plagiátorství [16].

- „doslovné opsání nebo kopírování cizího textu a jeho vydávání za vlastní, aniž by byl citován
- převzetí a publikování cizí práce (i seminární), včetně té, která ještě nebyla dokončena a odevzdána
- vydávání kompilace (nebo její části) za vlastní originální text
- okopírování grafických prvků bez citace a odkazu na původní zdroj
- okopírování názvu, struktury (např. obsahu, osnovy aj.) cizího textu, popřípadě až do té míry, že je možná záměna obou děl (§ 45 Autorského zákona)
- úmyslné neuvedení některých využitých zdrojů
- koupení či stažení volně dostupné práce a její vydávání za vlastní“ [16]

Úmyslného plagiování se dopouští mnoho autorů, ať už studentů, vědeckých pracovníků či zaměstnanců. Je to jev, který vzrůstá hlavně díky snadné dostupnosti materiálů na Internetu, kde panuje značná anonymita a můžeme čerpat z rozličných pramenů. Nárůst plagiátorství se odehrává hlavně mezi studenty, kteří si takto usnadňují svou práci na škole. Masově vznikají tzv. „paper mills“, tedy servery, kde lze jednoduše koupit starší seminární nebo jinou práci, případně ji stáhnout zadarmo. Větším problémem je práce na zakázku, kdy se jiný autor za úplatu nechá najmout na vypracování např. závěrečné práce, dokazování je zde velmi obtížné i proto, že se může jednat o kvalitní práci a autorství originálu si nikdo nenárokuje (kromě kupujícího) [5].

Plagiátorství neúmyslné

Tato forma plagiování vzniká převážně na autorově straně z nedbalosti nebo z neznalosti citační etiky, kdy autor zapomene uvést všechny zdroje. Může se jednat o parafrázování bez uvedení pramene, pokládání přebírané myšlenky za obecnou znalost, „autoplagiátorství “ nebo kryptomnesii [5].

Autor se zde nedopouští cíleného nečestného jednání za účelem vytvoření neautorského díla, ale kvůli malým zkušenostem, nedostatečným znalostem nebo časovému tlaku, jedná v omylu jako plagiátor.

Formy neúmyslného plagiátorství

Nejběžnějšími formy tzv. nevědomého plagiátorství jsou [16]:

- Nedodržení citační etiky - Dopustit se neúmyslného plagiátorství je jednodušší zvláště při vypracovávání studentských prací. Studenti mohou přehlédnout formulace citační etiky, tedy popisu, který jednoznačně určuje zdroje převzatých či citovaných pasáží, ze kterých autor čerpal, a také díky kterým můžeme jednoduše dohledat původní originální zdroj.
- Nesprávná kompilace - Pokud se rozhodneme tvořit kompilaci, musíme dbát některých pravidel, zejména použití více zdrojů a vytvoření syntézy, dále pak nesmíme sestavit práci pouze z citovaných pasáží několika zdrojů, jelikož se nebude jednat o kompilaci, a samozřejmě musíme uvést všechny prameny, ze kterých jsme čerpali.
- Nesprávná parafráze - K pochybení při parafrázování dochází při neuvedení zdroje přepisované myšlenky nebo nedostatečném parafrázování.
- Nesprávné rozpoznání všeobecně známých faktů - Obecně známý fakt je obecnou myšlenkou, která nemusí mít uvedený zdroj. Může se jednat o základní matematické rovnice, historická fakta apod. Zde je třeba být pozorný a raději uvést zdroj v případech, kde si nejsme jistí.
- Využití vlastních děl („self-plagiátorství", „auto-plagiátorství") - Pokud se v naší práci opíráme o svou předešlou práci, musíme ji také uvést ve zdrojích. Jinak se jedná „auto-plagiátorství". Ikdyž autor vlastně nevykrádá cizí dílo, protože je jeho autorem, určitě porušuje citační etiku. Osobně bych považoval tento akt za nedbalost nebo pochybení, ale rozhodně ne za plagiátorství. Jiným případem je kauza Jaroslava Světlíka, děkana FMK Univerzity Tomáše Bati ve Zlíně. Ten byl usvědčen etickou komisí z plagiátorství tím, že opsal některé části své doktorandské práce do práce docentské [13]. Zde tedy nefiguruje špatná citace vlastní práce, ale úmyslné usnadnění intelektuální práce.
- Kryptomnézie („skrytá paměť") - Lidská paměť je složitý mechanismus a může nás samotné přivést k omylu. V tomto případě se jedná o použití myšlenky, kterou považujeme za vlastní, aniž je skutečně naše. Autor zkrátka zapomněl, že se s myšlenkou setkal v jiném kontextu, a pokládá ji za vlastní.

2.5 Důvody k plagiátorství

Za každým plagiátem je nějaký motiv, neznalost či pochybení. Z pohledu akademického a komerčního plagiátorství se může těchto důvodů objevit několik [14].

2.5.1 Motiv k plagiátorství v komerční sféře

Narozdíl od akademického plagiátorství zahrnuje tato sféra nejrozličnější případy a důvody, které vedou autory k plagiarizování cizích děl. Může se jednat prakticky o cokoliv,

co lze nějak využít. V politice například o proslov oponenta v jiné záležitosti, v literatuře o texty jiných literátů, grafické podoby výtvarných děl, jejich prvky, počítačové hry nebo publicistické články, atd. Motivem mohou být:

- Finanční motivace
- Kariérní postup
- Firemní výhody
- Brzký termín odevzdání, případně nedostatek času
- Nedosatatek zkušeností
- Slabá etika
- Přitáhnutí pozornosti

Plagiátorství v komerční sféře je velmi problematické. Je téměř jisté, že jediným řešením je právní žaloba poškozené strany, protože zde došlo k porušení autorského zákona. Dokazování je velmi náročné na čas, energii i finance, viz. kauza 2.8.

2.5.2 Motiv k plagiátorství ve školství

- Nestudijní typ
- Lenost
- Špatný časový management
- Příliš pracovního zatížení
- Neporozumění látce
- Osobnostní tlak

Dlle mého názoru je většina výše zmíněných motivů, proč studenti plagiarizují, pravdivých a přesných. Ale neměli bychom zapomínat, že vůbec fakt, že někdo plagiarizuje, spočívá v něm samotném a v jeho zásadách.

2.6 Zabránění plagiátorství

Lze říci, že prevencí plagiátorství je vyvarování se všech forem plagiování zmíněných výše. Měli bychom se držet pravidel citační etiky, uvádět všechny původní zdroje, ze kterých jsme čerpali. Převzaté myšlenky, shrnutí, výsledky a závěry, uvedené v naší práci musí odkazovat na dohledatelný a ověřitelný zdroj. To samé se týká tabulek, grafických prvků a schémat, stejně tak i přeložených pasáží z cizojazyčné literatury. Neměli bychom kopírovat reference z jiných prací a své dílo také nenabízet k volnému použití u služeb jako „paper mills“ [16].

Povědomí o postižitelnost plagiátorství, hlavně v akademické oblasti při vypracovávání seminárních a závěrečných prací studenty, a o systémech, které umožňují detekci plagiátů, je další důležitý faktor v prevenci a odhalování plagiátorství.

2.7 Detekce plagiátů v přirozeném jazyce

S rozkvětem Internetu se významně rozšířil přístup k elektronickým dokumentům a zdrojům. Na tento fakt se můžeme dívat ze dvou hledisek. Jedno je snadný přístup k hodnotným informacím ze všech oborů, kdy v krátkém čase získáme potřebné studie, např. pro náš výzkum. Druhý pohled tkví v možnostech plagiátora, kde může čerpat z nepřeberného množství zdrojů a znesnadnit tím odhalení svého prohřešku.

Zjišťování plagiátů v textových dokumentech je velmi náročný úkol. Některé poznatky přispívají k odhalení takových prací. Manuální porovnání se opírá o lidské posouzení prací a vyžaduje důkladné kontrolování mnoha dokumentů zároveň. V případě, že má např. cvičící posoudit desítky prací studentů, je tento úkol prakticky nemožný. K odhalení dochází většinou v částech dokumentu, kde je značný jazykový rozdíl autorova stylu psaní. Mohlo by se jednat o náhlou změnu konstrukce vět, náhlé použití jiné slovní zásoby, než jakou autor užíval doposud, o shodnou strukturu s jinou prací (počet stran, osnova), případně o gramatickou chybu v textu. V posledním případě, pokud by vyučující našel stejnou chybu i v jiné práci, je velmi pravděpodobné, že se jedná o plagiát, avšak toto posouzení není stoprocentní. Proto se využívají automatizované postupy a systémy pro detekci plagiátů v seminárních a závěrečných pracích [17].

V dnešní době je v České republice několik detekčních systémů, které ověřují originalitu psaných dokumentů a studentských prací. Uvádím některé nejpoužívanější z nich: theses.cz (dostupný z `www.theses.cz`) a odevzdej.cz (dostupný z `www.odevzdej.cz`), jakožto české zástupce online systémů pro detekci plagiátů v textových dokumentech, a jako zahraniční např.: ithenticate.com (dostupný z `www.ithenticate.com`) a turnitin.com (dostupný z `www.turnitin.com`).

2.8 Detekce plagiátorství v programovacím jazyce

Vycházíme-li z definice 2.1 o plagiátorství, můžeme ji převést do kontextu programovacího jazyka, která jednoduše popisuje softwarový plagiát jako kód, který byl „zkopírován a změněn“ podle Joy a kol. [4]. Rozšíření této myšlenky je podle Parkera a kol. [7] plagiovaný program, který je definován jako „program, který byl vytvořen z jiného programu s malou mírou transformací. Běžné transformace, nejčastěji textové záměny, nevyžadují detailní porozumění programu.“

Definování plagiátorství, hlavně na vysokých školách, však nemá pevně stanovené hranice, kdy můžemě říct, že jde o plagiátorství, o porušení pravidel školy nebo jestli jde o provinění vůbec. Velmi zajímavou studií, která se zabývá definováním plagiátorství

ve výuce předmětů počítačových věd, je článek autorů [4], kteří vyhodnocovali dotazník o plagátorství mezi studenty, zodpovídaný profesory počítačových kurzů britských univerzit.

Nástroje a metodologie, které umožňují detekci plagiátů v softwarových artefaktech, podbrobně rozepisuji v kapitole 3.3.

Kauza SCO a Linux

Příkladem ze světa softwaru, kdy se jedna ze stran měla dopustit plagiátorství, a tím mělo dojít porušení licenční dohody, je kauza obecně označována za „SCO vs. Linux".

Vše začalo žalobou společnosti The SCO Group o porušení licenční dohody firmou IBM, která měla umožnit volný přístup k autorsky chráněnému vlastnictví, konkrétně k systému UNIX a jeho částem. Pro pochopení celé kauzy, která je komplikovaným příkladem právního boje mezi společnostmi o proprietární a open source řešeními, některými označována jako útok na samotnou open source myšlenku, musím uvést mnoho historických faktů, které předchází samotné kauze.

IBM podepsala v roce 1985 licenční dohodu s AT&T o použití UNIXu k tvorbě vlastního systému AIX a zavázala se k nezveřejnění jakýchkoliv částí kódu. V roce 1993 společnost AT&T prodává autorská práva k UNIXu (zahrnující UNIX System V verze 4, zk. SVR4 a další) společnosti Novell. Ta pokračuje ve vývoji, vytvoří vlastní verzi systému: UnixWare, sloučením NetWaru a SVR4. V roce 1995 se Novell rozhodne prodat práva k UNIXu, zahrnující zdrojový kód, dokumentaci, související běžící kontrakty spojené s provozem zakoupených unixových systémů, licence a intelektuální vlastnictví společnosti Santa Cruz Operation. Tato prodejní smlouva specificky nezahrnuje copyright a později je v dodatku ke smlouvě z roku 1996 doplněno, že nezahrnuje copyright s výjimkou „práv a obchodních značek, vlastněných Novellem, k datu kontraktu, které jsou vyžadovány spol. Santa Cruz Operation k uplatnění jejích práv vzhledem k akvizici UNIXu a technologii Unixwaru." [9]

Společnost v roce 2000 prodává aktiva a práva k UNIXu společnosti Caldera Systems, která se vzápětí přejmenuje na The SCO Group (dále pouze SCO). V této fázi vývoje události vyvěrá mnoho spekulací, proč se tak vlastně stalo, ale faktem zůstává, že v roce 2003 podala SCO podala několik žalob o práva k intelektuálnímu vlastnictví částí UNIXu a Linuxu. Linuxová a open-sourcová komunita, koncoví zákazníci, kteří užívající linuxové řešení a společnosti zabývající se vývojem těchto systému jsou ohroženi. Linux je derivát UNIXu s kompletně oddělenou a otevřenou kódovou databází, kdy jej v roce 1991 začal implementovat Linus Torvalds okolo GNU projektu Richarda Stallmana(sice vycházejícího z UNIXu, ale zbaveného všech proprietárních částí)[12].

Žaloba SCO o 1 miliardu dolarů se ze začátku zaměřuje na porušení licenční dohody s IBM, která, jak tvrdí SCO, porušila obchodní tajemství UNIXu tím, že zabudovala UNIXové intelektuální vlastnictví do Linuxu[12]. Přesněji, zkopírováním částí UNIXu "line-by-line"do systému Linux, někdy také pozměněním kódu, aby nešel rozpoznat od originálu. Toto tvrzení podemílá open-sourcovou filozofii, která zaštiťuje Linux a znamenala by teoretickou nutnost dokoupení licencí pro všechny společnosti používající Linux.

V roce 2005 se rozběhl soudní spor o práva k UNIXu mezi SCO a Novellem, vztahující se na předmět kontraktu z roku 1995 mezi Novellem a Santa Cruz Operation. SCO nárokovalo vlastnictví copyrightu a dalších intelektuálních vlastnických práv. Jako odpověď na hrozby ze strany SCO společnost Novell zveřejnila, že během začátku roku 2003 ji společnost SCO žádala o transfer práv k UNIXu. Tímto krokem de facto přiznala práva Novellu. Novell zamítl převést práva a obvinil SCO z nárokování práv, které ji nepatří.

10. srpna 2007 rozhodl federální okresní soud v Utahu ve věci vlastnictví copyrightu k UNIXu ve prospěch Novellu. Soudce federálního soudu také poznamenal, že Novell, jako oprávněný vlastník výlučných práv k UNIXu, může přinutit SCO, aby upustilo od žaloby nárokování pohledávek vůči IBM. Tímto rozhodnutím fakticky došlo k „potopení" žaloby SCO z roku 2003 proti IBM [10]. Novell oznámil, že „nemá zájem o souzení uživatelů UNIXu", a „nevěříme, že v Linuxu je Unix."[11]

Tato kontroverzní kauza o práva k UNIXu ukázala, že SCO nenašlo žádná porušení copyrightu v Linuxu, ačkoliv měli přístup k celé kódové databázi [11]. Pro linuxovou komunitu to znamená vítězství nad SCO, tento příklad však ukázal, jak dlouhé a nejisté mohou být soudní spory o vlastnictví softwaru. Ačkoliv bylo vyvráceno, že šlo o plagiátorství či porušení intelektuálního vlastnictví chráněného patenty a copyrightem, je toto téma velmi aktuální a nelze jej ignorovat.

2.9 Shrnutí

V této kapitole byly rozebrány pojmy z oblasti plagiátorství, jeho formy, výskyt a motivy. Byly zmíněny některé případy plagiátorství z akademické i komerční sféry. Ačkoli je automatizace již zcela běžnou praxí při vyhledávání plagiátů a také nezbytnou částí kontroly jakýchkoliv odborných či jiných prací, pořád zůstává konečný verdikt na lidském zhodnocení. Úrovně softwarového plagiátorství a metodologie sloužící k jeho odhalení jsou popsány v následující kapitole.

3 Detekční techniky a algoritmy

V této části práce se věnuji detekčním technikám a algoritmům, které se používají při vyhledávání plagiátů ve zdrojovém kódu. Na úvod musím poznamenat, že ačkoliv tyto nástroje dosahují směrodatných výsledků při určování míry podobnosti mezi dvěma entitami (v kontextu této práce budu tímto pojmem označovat objekt předložený k detekci, ať už textový nebo softwarový), nelze se spoléhat pouze na ně. Tyto nástroje poskytují pouze pravděpodobnostní ohodnocení, zdali je zkoumaná entita plagiátem, avšak finální rozhodnutí náleží manuálnímu přezkoumání.

Velmi podobnou doménou, jakou je detekce plagiátů, je i vyhledávání kódových klonů v softwarových systémech za účelem zefektivnění běhu programu nebo zvýšení výkonu. Metodologie detekce klonů je stejná i pro detekci plagiátů, a tudíž ji můžeme použít v kontextu našeho tématu [18].

Nejčastěji se detekční systémy využívají u ověřování prací studentů v předmětech počítačových věd.

3.1 Úroveň modifikace zdrojového kódu

Jak jsem uvedl v kapitole 2.8, plagiovaný program prochází modifikací ze strany plagiátora, aby jej mohl vydávat za vlastní. Modifikace, které mění podobu kódu, mohou mít rozsah několika kategorizovaných úrovní, definovaných podle Clougha [15] od nejjednodušších úprav formátování a změny názvu proměnných, přes umísťování nadbytečných výrazů neovlivňujících běh programu, až po ty komplexní, kdy plagiátor mění kontrolní struktury programu (záměny cyklů `for` a `while`, apod.). V literatuře je charakterizováno 7 úrovní modifikace kódu [7]. Příklady úrovní modifikací jsou zobrazený na obr. 18.

Úroveň 0 znamená zkopírování kódu bez jakýchkoliv úprav. Takový plagiát je nejjednodušší odhalit, jelikož je věrnou kopií originálu a nevyžaduje žádné programovací znalosti.

Úroveň L1 zahrnuje úpravu komentářů a formátování kódu. Jedná se o vizuální změny, jako je úprava prázdných řádků, tabulátorů a mezer. Komentáře jsou parafrázovány, přidány, přeloženy, nebo naopak úplně odstraněny [17].

Úroveň L2 je změna názvů identifikátorů. Jedná se o přejmenování proměnných nebo metod. Tato úroveň plagiování nevyžaduje žádnou znalost programování a je stejně jako předchozí úrovně lehce manuálně odhalitelná v krátkých zdrojových kódech.

Modifikace na úrovni L3 zahrnuje úpravu deklarace proměnných. Může se jednat o změnu datového typu, umístění deklarace proměnné, přehození pořadí operandů v příkazu bez ovlivnění výsledku nebo záměnu globálních a lokálních proměnných.

L4 úroveň se týká změny pořadí metod nebo funkcí. Plagiátor může pozměnit zdrojový kód nahrazením volání metody implementací jejího těla a naopak, sloučit více metod do jedné. To samé platí o třídách a souborech, které program používá. Vzniká tak vizuálně jiný program, ale obsahuje stejnou funkcionalitu. Tato úroveň vyžaduje programovací znalosti a je hůře detekovatelná.

Úroveň L5 se týká významnějších zásahů do původního kódu, které ovlivňují vizuální a syntaktickou strukturu kódu. Jedná se o vkládání nadbytečných příkazů a metod, které v zásadě nemění funkcionalitu kódu, přepisování syntaktických struktur selekce a iterace nebo vkládání nedosažitelného kódu. Odstranění některých částí kódu, který není nezbytný, také mění podobu celého programu. Tyto zásahy vyžadují použití důkladnějších detekčních technik a vyšší míru abstraktizace při hledání podobnosti. Plagiátor takovými zásahy rozumí funkcionalitě programu, na jehož základě tvoří plagiát.

Poslední úroveň (L6) mění kompletně syntaktickou strukturu některých částí nebo celého programu. Sémantika je zachována, a tedy i konzistence původní logiky programu. Pod takovou záměnou si můžeme představit místo použití rekurzivní funkce faktoriál, posloupnost příkazů s cyklem `for`, implementovaných ve funkci main. Plagiátor rozumí funkcionalitě a ovládá teorii v pozadí daného programu. Názorná ukázka na obr. 18.

V novější literatuře se kategorizace modifikací zdrojového kódu rozebírají hlouběji, např. na lexikální a syntaktickou modifikaci s členitějšími úrovněmi, příkladem budiž Gasevic a kol. [19]. Avšak pro účely této práci stačí základní rozdělení.

3.2 Proces detekce plagiátů ve zdrojovém kódu

Abychom mohli najít podobnosti mezi dvěma entitami, musíme porovnat jednotlivé jednotky každé entity se všemi z entity druhé. Toto je časově i prostorově náročný úkol. Je tedy nutné předzpracovat kód každé entity, odstranit ty části, které nejsou nutné pro hledání podobnosti, a převést vstupní zdrojové kódy do podoby, která je lépe zpracovatelná pro detekční systém.

Tento proces vyhledávání plagiátů sestává z několik podkroků, které postupně upravují zdrojový kód do intermediálních prvků, které jsou pak převedeny do podoby, jež zpracuje algoritmus pro detekci, dále pak počítání podobnosti, seskupení výsledků a vizualizace. Proces je graficky jednoduše znázorněn na obr. 2. Jedná se o více méně obecný postup, který je používán pro vyhledávání klonů v soft. systémech a lze jej aplikovat i pro detekci plagiátů. Nejedná o se striktní univerzální postup. Některé fáze se mohou lišit v závislosti na použité metodologii. Uvádím jej pro ucelení představy, jak takový systém funguje, a vycházím převážně z práce o detekci klonů od Roye a Cordyho [2].

```java
public class App {

    static int faktorial(int n) {
        int a = 1;
        for (int i = 1; i <= n; i++)
            a = a * i;
        return a;
    }

    static double deleni(int a, int b) {
        if (b == 0)
            System.out.println("Nelze delit nulou");
        else
            return (double)a / (double)b;
        return 0;
    }

    static void vypis(int n) {
        if (n < 10)
            System.out.println("Mensi: " + n);
        else
            System.out.println("Vetsi: " + n);
    }

    public static void main(String[] args) {
        int number;
        int VALUE = number = 5;
        int a;
        int delitel = 7;
        double d;

        a = faktorial(VALUE);
        vypis(a);

        d = deleni(a,delitel);
        System.out.println("Vysledek deleni: " + d);

        for(int x = 1;x<10;x++)
        {
            number = number * x;
        }
    }
}
```

```java
public class Application {
    //L3 misto tridni promenne deklarujeme globalni
    static final int DELITEL = 7;
    /*
     * L3 - zmena nazvu metody oproti puvodni, zmena podminky
     * L4 - zmena poradi umisteni metody
     * L1 - zmena formatovani kodu + pridany komentare
     */
    public static void print(int n) {
        if (n > 10)
        {
            System.out.println("Vetsi nez 10: " + n);
        }
        else
        {
            System.out.println("Mensi nez 10: " + n);
        }
    }
    /*
     * L6 - syntakticka zmena struktury tela metody,
     * vyuziti rekurze oproti
     * iteraci, semantika je stejna
     */
    static int faktorial(int n) {
        if (n == 0)
            return 1;
        else return n * faktorial(n - 1);
    }
    public static void main(String[] args) {
        //L2 a L3 - umisteni deklaraci, zmena nazvu promennych,
        //datovych typu
        int a,n,VALUE = 5;
        n = VALUE; float d;
        a = faktorial(VALUE);
        print(a);
        //L4 - misto volani metody vlozeni jejiho tela
        if((DELITEL)!=0) {d = (float)a/(float)DELITEL;}
        else {
            System.out.println("Nelze delit nulou");
            d = 0;}
        System.out.println("Vysledek deleni: " + d);
        //L5 - zmena kontrolni struktury cyklu
        int x = 1;
        while(x<10) {
            n = n * x;
            x++;}
    }
}
```

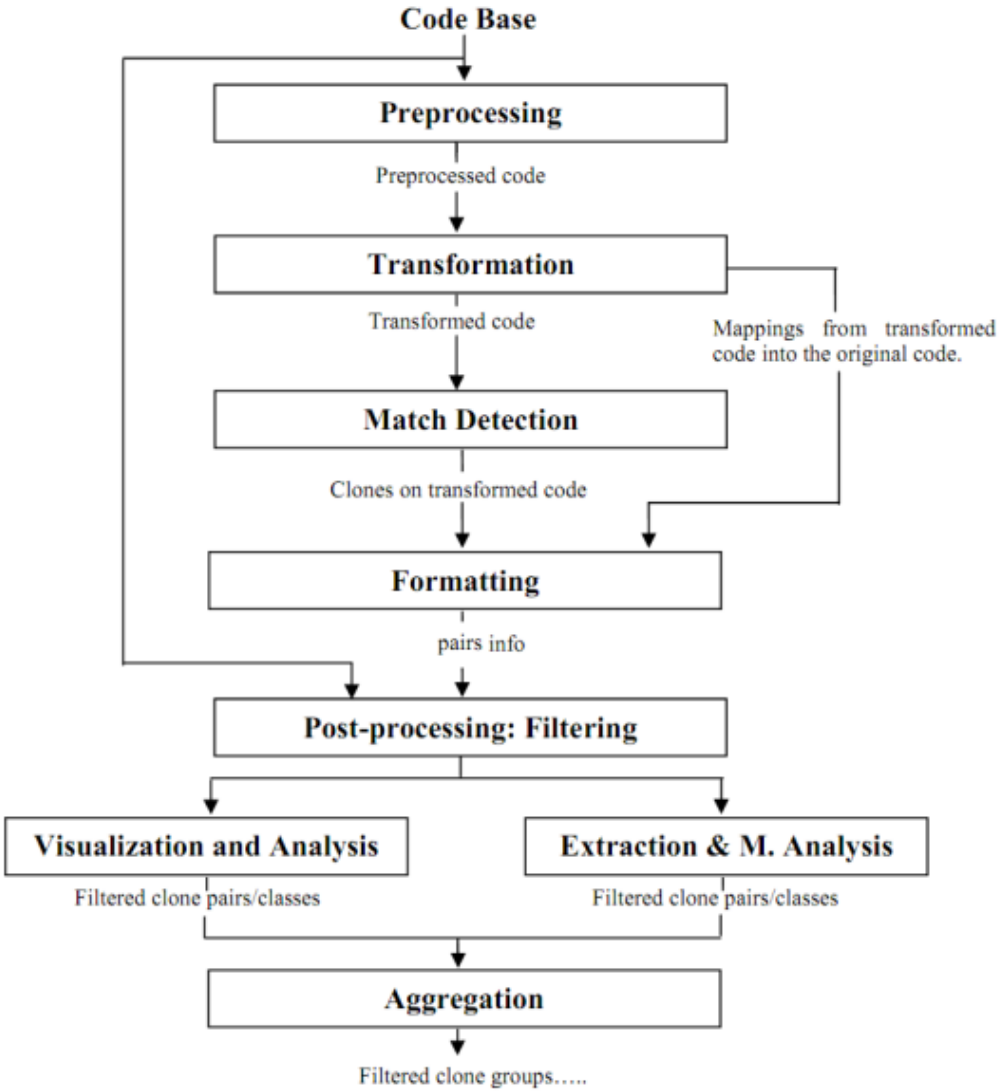
Obrázek 1: Ukázka modifikací v Javě

3.2.1 Předzpracování (Preprocessing)

V první části se zdrojové kódy entit rozdělí do patřičných domén, ve kterých bude probíhat porovnávání. Jedná se o filtrování souborů, které nejsou pro srovnání zajímavé, a další základní úpravy v těchto souborech.

V první řade se odeberou automaticky vygenerované části kódu, zabudované SQL dotazy, kód vygenerovaný z jiných nástrojů (jako JFlex, YACC apod.) nebo se může jednat o direktivy preprocesoru v C++.

Dalším krokem v této fázi je rozdělení souborů se zdrojovými kódy do disjunktních množin - zdrojových jednotek, tyto základní částice se přímo účastní fáze porovnávání. Tyto jednotky jsou vytvořeny ze souborů, tříd, metod, bloků kódů, výrazů nebo sekvence jednotlivých řádků apod. Takové zdrojové jednotky můžeme rozčlenit na menší prvky



Obrázek 2: Proces detekce plagiátu [2].

podle lexikální analýzy (na textové řetězce nebo tokeny) a podle syntaktické analýzy (na uzly syntaktického stromu). Toto dělení spadá pod další fázi procesu detekce.

3.2.2 Transformace (Transformation)

- *„Pretty - Printing“* - Jedná se o stylistickou úpravu textu, zdrojového kódu nebo značkovacího jazyka. Používá se stylistické formátování pro lepší čitelnost obsahu, například zvýraznění syntaxe.

- *Odstranění komentářů* - Většina detekčních přístupů odstraňuje nebo ignoruje komentáře, protože je plagiátor může lehce pozměnit. V technikách, které měří podobnost podle metriky, se komentáře zahrnují do počítání podobnosti.
- *Odstranění mezer (whitespace)* - Mezery se v drtivé většině detekčních přístupů nepovažují za atributy k hledání podobnosti, nejjednodušší úpravy plagiovaného kódu jsou změny ve formátování. V přístupech, které porovnávají řetězce řádků, se počítá s řádkovým zalomením.
- *Tokenizace* - Metodologie, které pracují se sekvencí tokenů, pracují tak, že vezmou každý řádek zdrojového kódu a podle lexikálního pravidla konkrétního programovacího jazyka jej rozdělí na tokeny. Takto se ze souborů a tříd sestaví sekvence tokenů, které vstupují do fáze detekce shody. Ze sekvence jsou odstraněny komentáře, mezery, řádková zalomení a tabulátory.
- *Parsování* - Syntaktická analýza vstupních dat (řetězců nebo tokenů) na základě formální gramatiky umožňuje vznik parsovacího stromu nebo abstraktního syntaktického stromu (AST). Jednotlivé podstromy pak tvoří zdrojové jednotky určené k porovnání.
- *Generování grafu (PDG - Program dependence graph)* - Přístup, který zohledňuje sémantický význam informací ve zdrojovém kódu s vysokou abstraktizací. Zahrnuje kontrolní a datový tok programu v izomorfních grafech, v jejíchž podgrafech hledá podobnosti. Některé metrické přístupy mohou využívat PDG.
- *Normalizace identifikátorů* - Je aplikována ve většině přístupů, nahrazuje proměnné a jiné identifikátory za jeden typ tokenů nebo symbolů.
- *Transformace programových prvků* - Dalším stupněm úprav je nahrazení syntaktického prvku za jiný, obecnější. Např. cyklus `for`, `do – while` nebo `while` může být nahrazen symbolem `loop`.
- *Počítání metrických hodnot* - Přístupy, porovnávající metriky zdrojových kódů, počítají některé atributy z neupravených nebo transformovaných zdrojových kódů a jsou vyhodnoceny v další fázi.

3.2.3 Detekce shody (Match detection)

Do fáze detekce shody vstupují předzpracované a transformované jednotky, vhodné pro algoritmus, který provádí srovnání jednotek ze vstupních entit, každá s každou (každá z první entity s každou z druhé entity). Výstupem z této fáze je seznam nalezených shod vzhledem k transformovanému kódu. Shody by měly být spárovány ke svému protějšku každé z entit. Výstupní informace z této fáze se vztahují k částem kódu, kde došlo ke shodě z 1. a z 2. entity podle srovnávacího algoritmu. Mohlo by se například jednat o čtveřici (1. Entita: Začátek, 1. Entita: Konec a 2. Entita: Začátek, 2. Entita: Konec).

Algoritmy, které se využívají pro detekci podobnosti budou rozvedeny v 3.4 (využívají se např. Suffix-tree, Dynamic pattern matching, hash-value comparison, greedy-string-tiling, atd.).

3.2.4 Formátování (Formatting)

Seznamy shod s ohledem na transformované jednotky, se kterými se pracovalo v části předchozí, se v této části konvertují na kolekce shodných párů namapovaných na původní zdrojové kódy entit. Každá lokace shody odkazuje na konkrétní soubory a řádky v původních zdrojových souborech. Nalezenou dvojici nebo pár s vysokou mírou podobnosti si můžeme představit jako dvojici z množiny shod: {(E1: Název souboru, E1: Začátek shody, E1: Konec shody),(E2: Název souboru, E2: Začátek shody, E2: Konec shody)}. E1 reprezentuje první srovnávanou entitu a E2 druhou. Nelze rozhodnout, jestli je první entita plagiát z druhé nebo naopak.

3.2.5 Závěrečné zpracování a vizualizace (Postprocessing and Vizualization)

V poslední fázi detekčního procesu se výsledky zobrazené v původním kódu filtrují manuálně. Takto můžeme snížit počet falešně pozitivních detekovaných plagiátů. Tato část také zahrnuje vizualizaci výsledků. Může se jednat o textový soubor s odkazy na příslušné podezřelé dvojice nebo automaticky vygenerovaný report ve formátu HTML. Některá řešení výstupu jsou rozebrána v kapitole 7.

Agregace se využívá u detekce klonů. Nalezené shody se agregují od nejnižších detekovaných jednotek až po nejobecnější (od klonů nejmenších po třídy klonů nebo clustery).

3.3 Detekční techniky a postupy

Detekční technika je v podstatě sekvence procesů, která na základě typu přístupu, zpracovává vstupní data a vyhledává plagiáty na základě podobnosti. Z obecného pohledu pracují tyto techniky ve dvou fázích (ikdyž prakticky z více). Jedná se o transformaci a porovnání. V první fázi je zdrojový text transformován do interního formátu, který umožňuje použití efektivnějších porovnávacích algoritmů. Ve druhé probíhá porovnávání, kdy se detekují shody. Vnitřní formát, který reprezentuje formu porovnávaných dat, poslouží jako klasifikační klíč, podle kterého popíšu jednotlivé kategorie detekčních technik [2].

3.3.1 Textový přístup

Tento typ technik je založen na metodách porovnávajících texty nebo řetězce. Vstupním formátem dat jsou sekvence řetězců a algoritmus ve většině případů vyhledává totožné podřetězce po řádcích. Výstupem vyhledávacího algoritmu je detekovaný pár podezřelých částí, které obsahují plagiovaný textový fragment v jeho maximálním rozsahu. Tento

Opearce	Jazykový symbol	Příklad	Nahrazení
1	Řetězcový literál	”Text”	”...”
2	Celé číslo	12	1
3	Desetinné číslo	12.3	1.0
4	Identifikátor	counter	i
5	Číselný dat. typ	int, byte, double	number
6	Název funkce	print()	method()

Tabulka 1: Normalizace, příklad zobecnění některých jazykových prvků [2].

přístup provádí malou měrou transformaci či normalizaci. Jelikož se jedná o textový, případně částečně lexikální přístup, nebere se v potaz strukturální stránka zpracovávaného kódu, a tedy tento přístup neodhaluje modifikace spadající do vyšších úrovní modifikace.

Některé používané formátovací procesy (pro vstupní zdrojové kódy ve fázi transformace) zvyšují čitelnost a vytváří uniformitu zpracovávaných vstupních textů, a tím zvyšují robustnost celého detekčního systému. Ačkoliv nesmíme zapomenout, že tento typ formátování je odolný oproti nižším úrovním modifikací.

- *Odstranění komenářů* - Všechny komentáře v programu jsou ignorovány. Tento typ formátování dělá vyhledávací algoritmus odolný vůči modifikaci komentářů, tedy úroveň L1 3.1.
- *Odstranění bílých mezer* (whitespace) - Odstranění tabulátorů, mezer a řádkových zalomení. Získává odolnost vůči změnám formátování v kódu.
- *Normalizace* - pro zvýšení odolnosti pro některým typů modifikací, jako je přejmenování názvů proměnných nebo změna datových typů, se používá částečná lexikální analýza - normalizace - kdy se některé textové prvky nahradí obecnou reprezentací konkrétní skupiny prvků - tokeny, viz. 1. tímto typem transformace získává systém odolnost vůči L2 úrovni a některým případům L3.

Příkladem, jak probíhá detekce plagiátů založená na textovém přístupu, je systém `DUP`. Ten detekuje plagiáty řádek po řádku použitím lexeru a algoritmu, který vyhledává řetězce podle tokenů jednotlivých řádků. Ve fázi transformace odstraní tabulátory, prázdné mezery a komentáře. Normalizuje některé řetězcové hodnoty jako názvy proměnných, funkcí a datové typy obecnějšími zástupci. Spojí všechny takto formátované řádky do jednoho řetězce hashovací funkcí, ohodnotí každý řádek pro srovnání a extrahuje množinu párů nejdelších shod použitím *suffix-tree* algoritmu.

Obdobné systémy mohou používat nejrůznější algoritmy pro hledání podobností, např. *Dynamic pattern matching*, algoritmus založený na vytváření otisků, tzv. *fingerprinting*, UNIXovou utilitu *diff*, výše zmíněný *suffix-tree* algoritmus nebo latentní sémantickou analýzu [2]. Příklad, jak vypadá kód před a po normalizaci, je na obr. 3.

```java
//metoda pro vytisteni matice
    public void printMatrix() {
        for (int i = 0; i < one.length(); i++) {
            if (i == 0) {
                for (int z = 0; z < two.length(); z++) {
                    if (z == 0) {
                        System.out.print("NEJAKY TEXT");
                    }
                    System.out.print(two.charAt(z) + " ");

                    if (z == two.length() - 1) {
                        System.out.println();
                    }
                }
            }
```

```java
publicvoidprintMatrix(){
for(inti=0;i<one.length();i++){
if(i==0){
for(intz=0;z<two.length();z++){
if(z==0){
System.out.print(");
}
System.out.print(two.charAt(z)+");
if(z==two.length()-1){
System.out.println();
}
}
}
```

Obrázek 3: Java kód před a po normalizaci

Poznámka 3.1 Normalizovaný kód je vytištěn i s řádkovým zalomením, ačkoliv při zpracování jsou jednotlivé zalomení ignorovány. Tento postup je tudíž odolný vůči modifikaci formátování originálního zdrojového kódu plagiátorem.

3.3.2 Lexikální přístup

Tento přístup zahrnuje porovnávání fragmentů na úrovni tokenů. Původní zdrojový kód prochází lexikální analýzou neboli tokenizací (to je proces velmi podobný lexikální analýze, kterou používají kompilátory programovacích jazyků), porovnávání je pak prováděno na podsekvencích tokenů. Tento přístup je odolnější vůči některým modifikacím narozdíl od textově-založených přístupů, zejména vůči formátování a přejmenovávání identifikátorů. Výstupem je dvojice plagiátů, korespondujících k původnímu originálnímu zdrojovému kódu vstupních entit [3].

Samotný proces tokenizace spočívá v konvertování sekvence znaků do odpovídajících tokenů. Lexikální analýza, kterou provádí skener, je součástí překladače zdrojového kódu. Každý programovací jazyk obsahuje skupinu určitých znaků, ze kterých se může skládat další struktury jazyka. V Javě jsou základními stavebními prvky tokeny, které zastupují jednotlivé prvky Javy, jako klíčová slova (`if`, `public`, `class`, ...), separátory ({, }, ;, ...), operátory (+, -, &&, %, atd.) a literály (čísla, řetězce, čísla s desetinnou čárkou), identifikátory (názvy funkcí, proměnných, tříd,..) atd. Z těch se poté skládají vyšší jazykové struktury: výrazy, příkazy, bloky/metody, třídy a balíčky. Skener (realizován jako konečný automat) zpracovává vstupní řetězce znaků až do nejdelší možné varianty a řetězec nahradí odpovídajícím tokenem (zpracovává vstup dokud nenarazí na znak, který už nepatří do skupiny slov, popisujících daný jazyk a pokračuje dalším řetězcem). Tento algoritmus používá přesně definovanou sadu instrukcí, vynechává komentáře, whitespace znaky, pracuje po jednotlivých řetězcích znaků po řádcích a tak odděluje jednotlivé tokeny. Na konci procesu posílá sekvenci tokenů do další části překladače [20].

Na podobném principu jako skener překladače funguje také proces tokenizace. Využívají se různé lexikální generátory jako JFlex [21], YACC, ANTLR apod., které můžeme integrovat do detekčních systémů.

V systémech pro vyhledávání plagiátů můžeme pak dále upravovat a generalizovat různé skupiny tokenů a zvyšovat efektivitu detekčních algoritmů. Vyšší míra abstraktizace dělá vyhledávání podobnosti odolnější vůči plagiátorským modifikacím. Můžeme nahrazovat různé datové typy, kontrolní struktury nebo identifikátory jedním typem tokenů, viz. obr. 4, kde vidíme reprezentaci různých programovacích prvků Javy jako sekvenci zástupných tokenů.

```
-IDENTIFIER-LPAREN-IDENTIFIER-PLUS-STRING_LITERAL-PLUS-IDENTIFIER-RPAREN-SE
-RETURN-IDENTIFIER-SEMICOLON-
-RBRACE-
-PUBLIC-VOID-IDENTIFIER-LPAREN-RPAREN-LBRACE-
-FOR-LPAREN-NUMBER-IDENTIFIER-ASSIGNMENT-NUMBER_LITERAL-SEMICOLON-IDENTIFIE
-IF-LPAREN-IDENTIFIER-EQUALITY-NUMBER_LITERAL-RPAREN-LBRACE-
-FOR-LPAREN-NUMBER-IDENTIFIER-ASSIGNMENT-NUMBER_LITERAL-SEMICOLON-IDENTIFIE
-IF-LPAREN-IDENTIFIER-EQUALITY-NUMBER_LITERAL-RPAREN-LBRACE-
-IDENTIFIER-LPAREN-STRING_LITERAL-RPAREN-SEMICOLON-
-RBRACE-
-IDENTIFIER-LPAREN-IDENTIFIER-LPAREN-IDENTIFIER-RPAREN-PLUS-STRING_LITERAL-
-IF-LPAREN-IDENTIFIER-EQUALITY-IDENTIFIER-LPAREN-RPAREN-MINUS-NUMBER_LITERA
-IDENTIFIER-LPAREN-RPAREN-SEMICOLON-
-RBRACE-
-RBRACE-
-RBRACE-
```

Obrázek 4: Ukázka tokenizovaného kódu získaného na základě fragmentu java kódu: 3.

Jelikož jsou v tomto přístupu zpracovány jednotky na lexikální úrovni - tokeny, nebere se v potaz syntaktická struktura kódu a nalezené podobné podsekvence mohou překrýt syntaktickou podobnost na vyšší úrovni [3]. Pokud bychom zvážili integraci metodologie, která by tento nedostatek odstranila, ať už ve fázi předzpracování nebo po detekci shody, lexikální přístup by byl robustnější a odolnější vůči vyšším úrovním modifikace. V kapitole 4 se na takový postup podíváme blíže.

Poznámka 3.2 Posloupnost tokenů je zobrazena po řádcích, výpis je namapován na původní kód. Řádková zalomení jsou při hledání podobnosti ignorována. Pro lepší vizualizaci jsou mezi jednotlivé typy tokenů umístěny „-“. Obrázek slouží pouze jako demonstrace, proto je zobrazen pouze výřez.

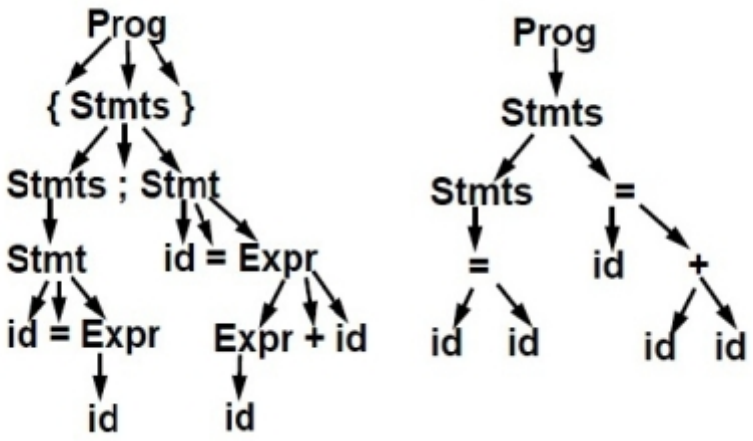
Některé nástroje pro detekci plagiátů používají algortimy pro hledání podřetězců v sekvencích tokenů. Můžeme použít některé techniky dolování dat nebo také Greedy-string-tiling nebo Smith-Watermanův algoritmus, které jsou rozebrány v další části.

3.3.3 Syntaktický přístup

Tento přístup pracuje se syntaktickou strukturou zdrojového kódu. Detekční systém používá parser ke konverzi programu do datové struktury parsovacího stromu nebo abstraktního syntaktického stromu (Abstract syntax tree - AST), kde se podobnost vyhledává na základě metriky blokových struktur nebo podobnosti podstromů [3].

AST je abstraktní reprezentace programu ve formě datové struktury hiearchického stromu. Představuje abstraktní syntaxi jazyka a zahrnuje pouze takové prvky jazyka, které ovlivňují sémantiku programu. Blokové struktury určující pořadí vykonávání příkazů (řekněme začátek a konec bloku) jsou odstraněny a příkazy popořadě začleněny do struktury stromu [22]. Listy zastupují terminály, tvoří je literály, konstanty a proměnné programu [17].

Na rozdíl od AST (viz. obr. 5) je parsovací strom tvořen všemi prvky syntaxe programovacího jazyku. AST reprezentuje tyto struktury právě ve svých uzlech a není potřeba u něj uchovávat všechny informace získané ze zdrojového kódu. Uzly parsovacího stromu představují neterminální pravidla bezkontextové gramatiky určitého programovacího jazyka a listy reprezentují syntaktické jednotky. Důsledkem této struktury parsovací strom obsahuje všechny tokeny vytvořené lexikální analýzou. Dále se tento typ stromu používá jako intermediální forma AST a je výpočetně méně náročný na produkci. V stromově založených technikách můžeme vyhledávat podobnosti na základě dvou metodologií.



Obrázek 5: Zleva: Parsovací strom, AST - rozdíl

Metody vyhledávání podobných podstromů

Tento přístup vyhledává plagiáty nacházením podobných podstromů. Jména proměnných, hodnoty literálů a ostatních prvků (listů ve stromu), které vznikly například toke-

nizací, jsou ve stromové reprezentaci abstraktizovány a dovolují tak efektivnější vyhledávání plagiátů.

Metrické metody

Pro všechny fragmenty kódu, jako jsou příkazy, bloky, metody nebo třídy, lze vypočítat jejich metriku. Při porovnávání se pak použijí tyto vektory místo srovnání přímo zdrojového kódu. Můžeme se jednat o techniky, které počítají otisky - *fingerprinting* nebo vzdálenosti mezi jednotlivými podstromy. Vytváření otisků jsou techniky, jejichž výpočty se pro jednotlivé fragmenty kódu vždy liší, jelikož funkce, které vypočítávají tyto hodnoty, jsou hashovací, a ty vytvoří pro byť minimálně pozměněné vstupy úplně odlišné výstupní hodnoty. Pak se porovnávají tyto metrické hodnoty a hledají se plagiáty mezi zpracovanými syntaktickými jednotkami. Ve většině přístupů se zdrojový kód převede na datové struktury parsovacího stromu nebo AST a pak se z nich počítají metriky. Musíme poznamenat, že počítání metrik se týká také PDG struktur, kde se pracuje s grafy a ne stromy.

3.3.4 Sémantický přístup

Přístupy, které berou v potaz i sémantiku programu a používají statickou analýzu, obsahují přesnější informace než syntaktické přístupy. Především se jedná o reprezentaci programu jako PDG (program dependency graph). Vrcholy představují výrazy nebo příkazy a hrany zastupují kontrolní a datové závislosti mezi vrcholy. Taková reprezentace se odpoutává od lexikálního pořadí, ve kterém jsou příkazy seřazeny a jeví se tedy z vnějšku jako sémanticky nezávislé [3]. Narozdíl od hran AST, kde sekvence příkazů ze zdrojového kódu není brána v potaz, je grafová prezentace robustnější proti plagiátorským modifikacím jako např. přehazování příkazů [22], změna řídících bloků nebo vkládání příkazů.

Tato technika dokáže odhalit mnohem vyšší úrovně modifikace kódu, ale je také mnohem náročnější na výpočet. Vstupní kódy se musejí převést na množinu grafů podle zrnitosti, se kterou se pracuje v transformační fázi, a pak se mezi nimi vyhledávají stejné izomorfní podgrafy. Takové postupy obsahují filtrování pro snížení počtu grafů [17].

3.3.5 Hybridní přístupy

Hybridní přístupy kombinují znaky lexikálních, syntaktických, případně sémantických přístupu. Takové kombinace se snaží získávat výhody použitých přístupů a zamezovat jejich nedostatkům tím, že se aplikuje jiný přístup. Jde tedy o kombinaci prezentace vnitřní podoby kódu (datových struktur) a použitých technik.

3.4 Algoritmy pro detekci plagiátů

Jednou z hlavní fází procesu vyhledávání plagiátů je detekce shody. Určování míry podobnosti mezi dvěma programy probíhá na základě porovnání podřetězců, podsekvencí tokenů, podstromů, na základě metriky nebo z podgrafů. Algoritmy, které tuto operaci provádí, můžeme někdy použít pro více přístupů, např. pro vyhledávání podřetězců i pro podsekvence tokenů. Některé z nich byly zmíněny v předchozí části podle typu přístupu. V této části popíšu ty nejběžnější, se kterými se můžeme setkat v praxi.

3.4.1 Greedy String Tiling

Greedy String Tiling (GST) je algoritmus navržený M. J. Wisem v roce 1993. Vznikl na základě pokusu a řešení problému porovnávání řetězců pro detekci plagiátů a porovávání sekvencí v DNA nebo proteinech. Algoritmus je detailně popsaný v článku M. Wise [23] a vycházím převážně z něj.

Původní systém navržený pro vyhledávání plagiátů se jmenuje YAP a pracuje ve dvou fázích. V první fázi se převádí text do sekvence tokenů, kterými nahrazuje textové fragmenty. Využívá přitom lexikální slovník slov, které náleží do domény řešeného problému (takže např. klíčová slova programovacího jazyka). Druhá fáze řeší určení míry podobnosti mezi dvěma řetězci tokenů s vysokou hodnotou indikující pravděpodobný plagiát. Algoritmus, který určuje míru podobnosti, sestává ze tří vlastností:

- Každý token v každém řetězci se může porovnat nejvýše jednou (pokud se shoduje s tokenem v druhém řetězci).
- Transponované podřetězce by měly mít minimální efekt na hodnotu podobnosti, protože transpozice se mohou vyskytovat pouze pokud nezpůsobují změnu sémantiky zdrojového textu. To znamená, že například změny v pořadí vykonávání příkazů `switch` nebo přehození příkazů v bloku kódu nemají vliv na výstup programu.
- Rozkládání složených příkazů by nemělo výrazně ovlivnit míru podobnosti snížením její hodnoty.

Dále se při vysvětlování GST algoritmus setkáme s několika definicemi podle [23]:

Definice 3.1 *Pokud se hovoří o dvou řetězcích, kratší se nazývá vzorový řetězec, zatímco delší je označen jako textový řetězec.*

Definice 3.2 *Maximální shoda (Maximal match) - je nejdelší sled shodných znaků mezi vzorovým podřetězcem P_p začínajících na indexu p a textovým podřetězcem T_t od indexu t. Shoda je nejdelší možná a pokračuje, dokud neskončí řetězec, nenastane neshoda nebo není jeden z prvků zaznamenán. Maximální shoda je označena jako* `max_match(p,t,s)`*, kde s je délka shody.*

Definice 3.3 *Dláždění (Tile) - je permanentní a unikátní (ve vztahu 1:1) asociace podřetězce z P a podřetězce z T. V procesu vytváření dláždění z maximální shody jsou jednotlivé tokeny z podřetězců zaznačeny a stávají se tak nedostupné pro další srovnávání. Dláždění délky s, začínající v P_p a T_t se označuje jako* `tile(p,t,s)`.

Definice 3.4 *Délka nejmenší shody (Minimum match length) - je to spodní hranice délky, která určuje, zdali budou vyřazeny ze srovnání všechny maximální shody (a tudíž i dláždění), pokud nemají délku větší než délku nejmenší shody. Tato hranice délky je minimálně 1, ale běžně se používá vyšší celé číslo.*

Předpokládejme, že je známa délka aktuální maximální shody (samozřejmě větší nebo rovna délce nejmenší shody) v algoritmu označena jako `maxmatch`, která je délkou největší možné shody, kterou ještě lze získat z P a T. Pseudokód GST pro porovnání dvou řetězců podle M. Wise 1: Algoritmus pracuje následovně. V první části se vyhle-

```
length_of_tokens_tiled := 0
Repeat
  maxmatch := minimum_match_length

  /* Tyka se neoznacenych tokenu podretezce P */
  for each Pp do
    /* Tyka se neoznacenych tokenu podretezce T */
    for each Tt do
      j := 0

      while (Pp + j = Tt + j AND unmarked(Pp + j) AND unmarked(Tt + j)) do
        j := j + 1

      /* Prida shodu do listu shod o delce j */
      if (j = maxmatch) then add match(p, t, j)

      /* Pokud je delka shody vetsi nez dosavadni, nahradi je novou */
      else if (j > maxmatch) then start new list with match(p, t, j) and maxmatch := j

  for each match(p, t, maxmatch) in list

    /* Vytvoreni noveho dlazdeni */
    if not occluded then
      for j := 0 to maxmatch − 1 do
        mark_token(Pp + j)
        mark_token(Tt + j)
      length_of_tokens_tiled := length_of_tokens_tiled + maxmatch;

Until maxmatch = minimum_match_length
```

Výpis 1: Pseudo kód Greedy String Tiling algoritmu [23]

dávají všechny nejdelší společné podřetězce ze vzorového a textového řetězce. Cykly prochází neoznačené tokeny z P a porovnává je s neoznačenými tokeny z T. Pokud narazí na shodu, tak nejvnořenější cyklus hledá nejdelší společné tokeny podřetězce. Pokud

se v nejvnořenějším cyklu narazí na řetězec, který má nejdelší společnou délku větší než největší v aktuálním seznamu shod - `maxmatch`, je tento seznam smazán a je přidán aktuální řetězec shody do `match(p, t, j)`.Takto je vytvořen seznam se všemi společnými podřetězci, mající délku alespoň rovnou délce nejmenší shody.

Ve druhé části probíhá vytváření dláždění nebo také označeování tokenů použitých pro dláždění. Tím se snižuje výpočetní délka. Pokud se některé dláždění překrývají, vybere se první z nich a ostatní zahodí. Cyklus `Repeat` probíhá, dokud se délky se aktuální délka nejdelšího shodného řetězce nerovná minimální délce shody, tak nedojde k zacyklení.

Výstupem je tedy seznam nejdelších společných podřetězců, které začínají na pozici p ve vzorovém řetězci P a ekvivalentně v textovém řetězci T s délkou s, díky těmto údajům můžeme výsledky vizualizovat a namapovat nalezené shodné řetězce k originálnímu zdrojovému kódu.

Pro vypočítání míry podobnosti můžeme použít vzorec podle práce [17]:

$$sim(P,T) = \frac{2 * coverage(tiles)}{|S| + |T|}$$

$$coverage(tiles) = \sum_{match(p,t,length) \in tiles} length$$

Tento hojně využívaný algoritmus má bohužel polynomiální složitost v nejhorším případě $O(n^3)$, kde n je počet tokenů v řetězci, pokud $|P| = |T| = n$ a v nejlepším případě se jedná o složitost $O(n^2)$ podle. Proto byly vytvořeny efektivní nástavby tohoto algoritmu, např. Karp-Rabinův GST a další variace, které jsou výkonnější a mají průměrnou složitost v $O(n)$. Nicméně v nejhorším případě zůstává tato složitost stejná jako u GST varianty [17].

3.4.2 Smith - Watermanův algoritmus

Lokální zarovnání je technika, která definuje podobné oblasti mezi dvěma sekvencemi, může se jednat o řetězce, nukleotidy DNA nebo sekvence aminokyselin v proteinech. Jedná se o snahu určení vztahů mezi jednotlivými oblastmi, kde se nachází nějaká podobnost (funkcionální nebo strukturální).

Smith - Watermanův algoritmus definovali tito pánové v roce 1981, aby vyhledával podobné oblasti mezi dvěma sekvencemi (právě v DNA nebo proteinech). Tento algoritmus využívá metody dynamického programování a směřuje k nalezení optimálního lokálního zarovnání vzhledem k použitému skórovacímu systému, který je složený ze skórovací matice (ohodnovacího schématu) a postihu za vkládání mezer. Taková kombinace metod konstruuje optimální řešení problémů z optimálních řešení jeho podproblémů a ukládá tyto mezivýsledky do tabulky, aby se předešlo opětovnému přepočítávání [1].

Obecně se lokální zarovnání zaměřuje na porovnání dvou lineárních sekvencí, aby byly nalezeny nejdelší podsekvence, které mají největší shodu. Ohodnocení (skóre) bloku sekvence je součtem jeho jednotlivých ohodnocení a hodnota optimálního zarovnání je hodnotou nejvýše ohodnoceného bloku sekvence.

Formálně máme dány dvě sekvence P a Q, kde p_i je prvkem P a q_j je prvkem Q tak, že $(i \leq |P|, j \leq |Q|)$, a kde je individuální ohodnocení $score(p_i, q_j)$ definováno jako:

$$score(p_i, q_j) = \begin{cases} m, & if\ p_i = q_i \\ d, & if\ p_i \neq q_i \end{cases}$$

kde m je ohodnocení pro shodu a d pro neshodu; m je hodnotou pozitivní a d negativní. Optimální hodnota zarovnání $Score(i, j)$ je vypočítaná na základě funkce skórovacího schématu:

$$Score(i,j) = max \begin{cases} Score(i-1, j-1) + score(p_i, q_i) & Match/Mismatch \\ Score(i-1, j) + g \Rightarrow w(a_i, -) & Deletion \\ Score(i, j-1) + g \Rightarrow w(-, b_j) & Insertion \\ 0 \end{cases}$$

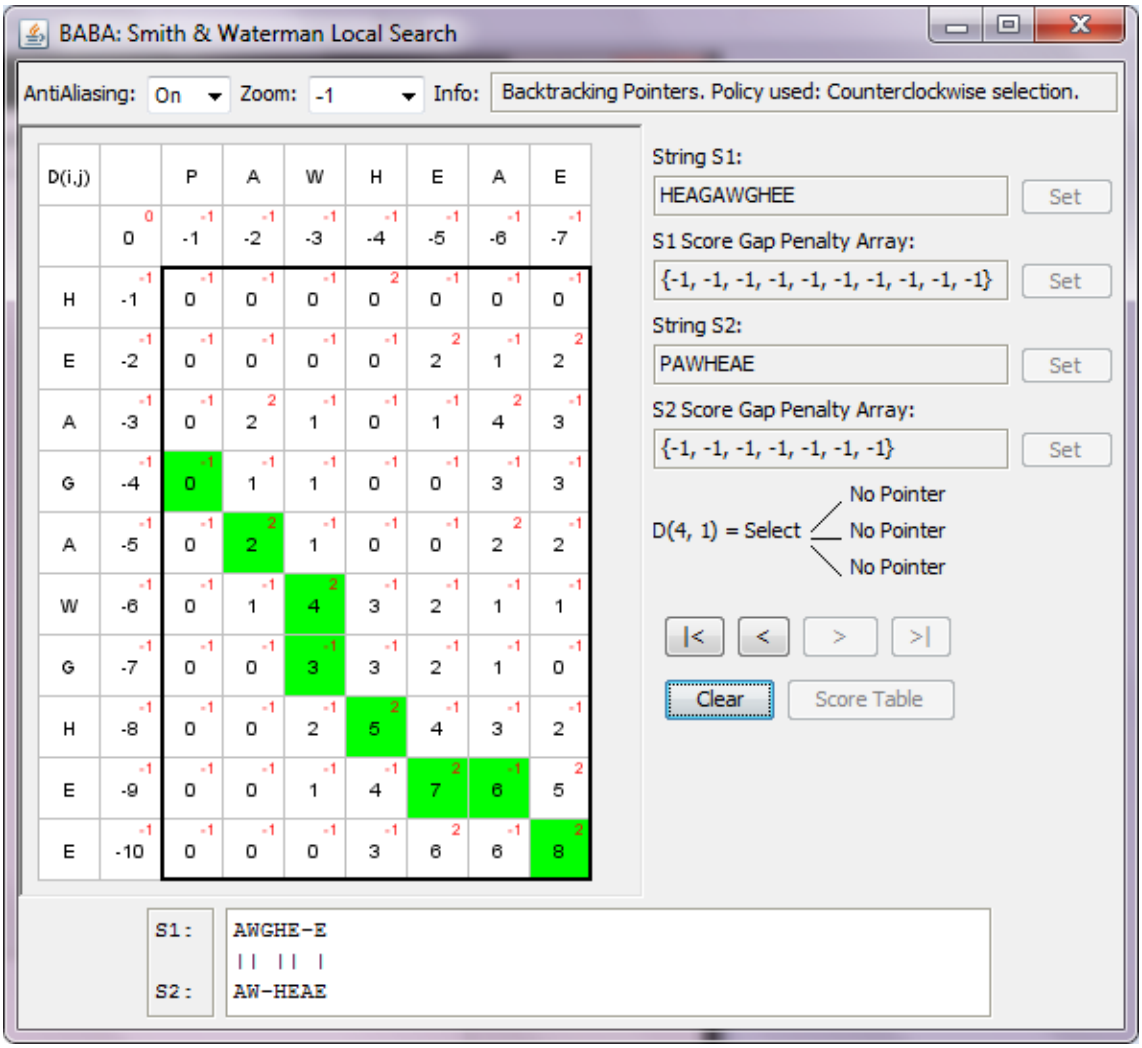
kde g je postih za shodu dík mezeře. Formálně řečeno, hodnota pro lokální zarovnání po hranici i a j je maximum jedné z možností: vložení mezery, shoda nebo neshoda, odstranění mezery a žádné srovnání. Takovým způsobem se vytvoří celá matice podobnosti a pro dva různé vstupní řetězce dostaneme výstup: obr. 6 .

```
SW classic --------------
The matrix H[][]:
      P  A  W  H  E  A  E
   0  0  0  0  0  0  0  0
H  0  0  0  0  2  1  0  0
E  0  0  0  0  1  4  3  2
A  0  0  2  1  0  3  6  5
G  0  0  1  1  0  2  5  5
A  0  0  2  1  0  1  4  4
W  0  0  1  4  3  2  3  3
G  0  0  0  3  3  2  2  2
H  0  0  0  2  5  4  3  2
E  0  0  0  1  4  7  6  5
E  0  0  0  0  3  6  6  8
Highest value in matrix H[][]=8
The value is in line 10,  colum 7
Sequence1 = HEAG-AWGHE-E
Sequence2 = ----PAW-HEAE
```

Obrázek 6: Matice podobnosti lokálního zarovnání

Backtracing neboli zpětné trasování utváří optimální lokální zarovnání z matice podobnosti. Tento proces vyhledá maximum v matici a od jeho pozice trasuje optimální

uspořádání jednotlivých prvků řetězců. Algoritmus trasování začíná v maximální hodnotě matice a zpětně zapisuje správné dvojice prvků, případně doplní mezery podle optimálnější hodnoty, tvořené při počítání matice. Tento proces lze vidět na obr. 7, převzatý z Java aplikace B.A.B.A.[1] od Casagrande, N., kde vidíme trasu zpětného uspořádání zarovnání mezi dvěma řetězci. V tomto případě autor použil ukazatele, které ukládal při vytváření matice. Při zpětném trasování tak od maximální hodnoty v matici pokračoval podle aktuálního ukazatele až k počátku optimálního zarovnání.



Obrázek 7: Backtracing - Optimální lokální zarovnání - trasování

[1]Casagrande, Norman. B.A.B.A.: Basic Algorithms of Bioinformatics Applet. Université de Montréal, 2003. Dostupné z: http://baba.sourceforge.net

Druhá možnost, jak může algoritmus trasovat, je vybírání původní hodnoty na jedné z pozic $H[i-1][j-1]$ (match/mismatch), $H[i][j-1]$(insertion) nebo $H[i-1][j]$(deletion), ze které byla vypočítána hodnota pro $H[i][j]$. Proces vytváření zarovnání se může zastavit u posledního ukazatele, pokud nenarazíme na prvek s hodnotou 0 nebo prvek na pozici $H[0][0]$ (v případě obr. 6 zpětné trasování probíhá do pozice (0,0)). První řádek a sloupec matice podobnosti obsahuje pouze prvky s hodnotou 0, abychom mohli určit skóre pro zarovnání prvního prvku prvního řetězce se všemi z řetězce druhého a obráceně.

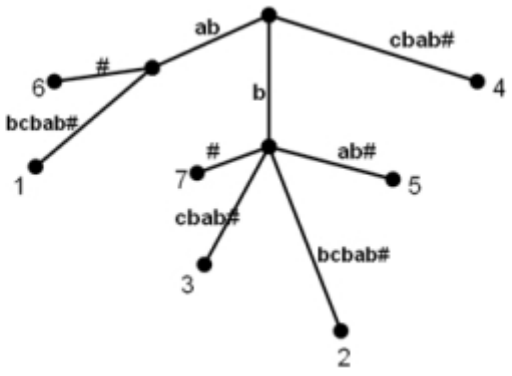
Díky tomu, že oba algoritmy pro výpočet lokálního zarovnání používají stejné ohodnocovací schéma, jsou výsledky stejné (s rozdílem, že zpětné trasování, jak vidíme na obr. 6, končí až v pozici (0,0)). Výstupem takového algoritmu je pak lokální zarovnání dvou řetězců s nejvyšším dosaženým ohodnocením (skóre), viz. spodní části výše zmíněných obrázků $Sequence1$ a $Sequence2$, respektive $S1$ a $S2$.

3.4.3 Suffix - tree Matching

Vyhledávání plagiátů na základě podobnosti podstromů je technika účinnější než lexikální přístupy, ale také je časově a prostorově náročnější.

V první řadě se podle daných pravidel vytvoří suffix tree struktura. Suffix tree pro řetězec T délky m je kořenový strom sestavený podle následujících pravidel:

- obsahuje přesně m listů, číslovaných od 1 až m
- každá hrana má popis podřetězce z T
- každý vnitřní uzel má alespoň dva potomky
- popisky dvou hran začínají ve vnitřním uzlu tak, že nezačínají stejným znakem
- popis cesty od kořene k listu je očíslován I je suffixem T, začínajícím v pozici i - $T[i..m]$



Obrázek 8: Příklad grafického znázornění suffix tree struktury [24].

Jeden z postupů používá pro srovnání a převod podstromů do AST Ukkonenovu verzi suffix - tree algoritmu. AST je prvně serializován podle jednotlivých uzlů stromu a také je vytvořen speciální atribut pro každý uzel, který určuje počet podstromů, jejichž je předchůdcem. Podstromy jsou dále serializovány podle pořadí, ve kterém se vyskytují v kódu, aby se zabránilo falešně pozitivním a falesně negativním nálezům plagiátu. Poté se pomocí suffix - tree algoritmu vytvoří suffix tree struktura AST a vyhledají se největší shody podstromů. Výsledky se zpracují tak, aby obsahovaly pouze syntaktické jednotky (plagiáty jsou obsaženy v celých blocích nebo příkazech). Takový postup zpracování stromové struktury má náročnost v $O(n + z)$, kde n je počet uzlů na vstupu a z je počet porovnání. Náročnost toho postupu je pak exponenciáoně nižší než jiné metrické techniky, které mají $O(n^2)$ [22].

3.5 Nástroje pro detekci plagiátů

V této podkapitole představím některé z mnoha nástrojů, které se snaží vyhledávat plagiáty ve zdrojových kódech [8].

Většina z nich aplikuje přístup transformace kódu do řetězce tokenů, kde tokeny jsou lexikální jednotky, ze kterých se budují bloky programu (identifikátory, klíčová slova, separátory, operátory, atd.). Na této úrovni se předpokládají změny od plagiátora, které nejsou sémantického rázu (modifikace komentářů, formátování a identifikátorů). V sekvenci tokenů se poté detekčním algoritmem hledá podobnost. Takto funguje většina nástrojů pro detekci plagiátů.

Další technikou, která se využívala hlavně v prvních detektorech plagiátů, byl přístup založený na počítání atributů. Pro každý soubor zdrojového kódu se počítalo několik vlastností, atributů, které charakterizovaly tento soubor. Pokud některé soubory měly větší podobnost, byly podezřelé na plagiátorství.

Novější možností, jak vyhledávat podobnosti, byly systémy zkoumající strukturu programu. Mohlo se jednat o porovnávaní na základě shodnosti podstromů v abstraktním syntaktickém stromu, nebo hledání podobnosti podgrafů v PDG [8].

Abychom mohli porovnat nástroje z více pohledů, uvedeme si některé vlastnosti, které pro nás mohou být zajímavé. V první řadě se jedná o vlastnost, se kterými programovacími jazyky dokáže systém pracovat. Další je rozšiřitelnost, kdy lze manuálně přidat rozpoznávání syntaxe dalšího jazyku. Dalším zajímavým parametrem detekčního systému bude styl prezentace výsledků, tedy vizualizace. Například zobrazení podobných částí kódů, zobrazených vedle sebe v barevném rozlišení, a sumarizované statistiky. Použitelnost znamená, jak je systém náročný na použití pro uživatele. Možnost vyloučení některých souborů ze srovnání. Například u kontroly studentských projektů, které vycházejí z nějaké základního projektu, je další vlastnost možnost vyloučit malé soubory. Může se dále jednat o webovou nebo lokální službu. A poslední vlastností je, zdali je detekční systém open - sourcem.

JPlag

JPLag byl vyvinut Guidem Malpohlem na Karlsruhské univerzitě v roce 1996 jako studentský výzkumný projekt. Nyní je JPlag webovou službou.

JPlag konvertuje program do řetězce tokenům, které reprezentují strukturu programu. Poté použije upravený „Greedy-string-tiling“ algoritmus pro porovnání dvou řetězců tokenů.

Marble

Marble je systém vyvinutý na univerzitě v Utrechtu v roce 2002 pro vyhledávání plagiátů v Java projektech.

Marble rozděluje vstupní program na základní soubory tak, aby každý obsahoval třídu nejvyšší úrovně. Poté je provedena základní transformace, která odstraňuje formátování a prvky, které se snadno mění (komentáře, importy, whitespacy, atd.). Pak je provedena lexikální analýza použitím regulárních výrazů a klíčová slova a další prvky jsou nahrazeny svými obecnými zástupci. Marble provádí dále třízení takto upravených souborů do dvou normalizovaných verzí. Porovnávání se provádí pomocí UNIXové utility `diff`.

MOSS

MOSS je zkratka pro „Measure Of Software Similarity“ a byl vyvinut na Standfordské univerzitě. Je provozován jako webová služba a je přístupný použitím skriptu.

MOSS měří podobnost na základě standardizovaných verzí dokumentů, používá tzv. otisky dokumentu pomocí metody zvané `winnowing`. Používaní otisků v dokumentu se provádí tak, že dokument je rozdělen do několika styčných podřetězců: k-gramů. Každý takový k-gram je hashovací funkcí ohodnocen a podmnožina všech hashovaných k-gramů je vybrána jako otisk dokumentu. Winnowing pak vyhledává mezi těmito množinami podobnost.

Plaggie

Plaggie je podobný JPlagu, akorát se používá lokálně a je open-sourcovým řešením. Vznikl v roce 2002 na Helsinské univerzitě.

Stejně jako JPlag používá ve fázi transformace tokenizaci a pro vyhledávání podobnostní slouží „Greedy string tiling“ algoritmus. Záměrně však autoři nepoužili úpravu algoritmu jako u JPlagu.

SIM

SIM je multi-jazykový systém pro detekci plagiátů. Pochází z roku 1989 univerzity v Amsterodamu. Nejnovější verze je z roku 2008.

Zpracování probíhá následovně. Prvně se tokenizuje vstupní program na sekvenci tokenů, pak systém vytvoří referenční tabulku, která slouží pro nalezení největší shody

Vlastnost/Nástroj	JPlag	Marble	MOSS	Plaggie	SIM
Počet prog. jazyků	6	1	23	1	5
Rozšířitelnost	ne	ne	ne	ne	ano
Vizualizace	5	3	4	4	2
Použitelnost	5	2	4	3	2
Vyloučení templatu	ano	ne	ano	ano	ne
Vyloučení souborů	ano	ano	ano	ne	ne
Lokální/webový	webový	lokální	webový	lokální	lokální
Open source	ne	ne	ne	ano	ano

Tabulka 2: Nástroje pro detekci plagiátů

mezi nově přidanými testovanými entitami, kterými mohou být zdrojové kódy, ale i textové dokumenty.

3.6 Shrnutí

V této kapitole jsem rozebral úrovně plagiátorských modifikací ve zdrojových kódech. Následoval popis obecného procesu detekce plagiátů. Z jakých částí se skládá a v jakém pořadí se provádí. Dále byly popsány detekční techniky a postupy, které se aplikují v oblasti detekce plagiátorství. Některé jsou používané na zdrojové kódy a jiné i na textové dokumenty. V závěru jsou detailně popsány tři algoritmy, které vyhledávají podobnosti, jedná se o GST, SW a Suffix - tree algoritmy. V úplném závěru popisuji nástroje pro detekci plagiátů.

4 Adaptivní lokální zarovnání klíčových slov

Systém pro detekci plagiátů ve zdrojových kódech s použitím adaptivního lokálního zarovnání klíčových slov byl navržen Jin-Su Limem a kol. [1] jako nová metoda pro detekci plagiátorství zdrojových kódů mezi velkými bázemi programů. Známé algoritmy pro detekci plagiátů jsou Greedy String Tiling nebo lokální zarovnání, zmiňované v části 3.4. Adaptivní lokální zarovnání (dále jen ALA) je variantou lokálního zarovnání s využitím matice podobnosti.

Tento přístup pro detekci plagiátů zkoumá podobnosti dvou zdrojových kódů, ale můžeme je také použít pro hledání podobnosti v bytekódu. Důvodem, proč hledat podobnosti na dvou různých úrovních (nebo čistě na úrovni strojového kódu), je dle mého názoru zaprvé samotná struktura bytekódu, která reflektuje podobu původního zdrojového kódu (napsaného v Javě) s tím, že některé plagiátorské modifikace jsou díky struktuře bytekódu neúčinné, jako formátování, editace komentářů nebo vkládání nesmyslných příkazů a zadruhé proto, aby původní zdrojové kódy nebyly „odkryty“ a díky zkoumání v detekčním systému je někdo nezneužil [6].

Dalším důvodem pro použití detekce na strojový kód je fakt, že komerční, ale i open sourcové (někdy) projekty neposkytují původní zdrojový kód, ale pouze zkompilované kódy. V takovýchto případech je jediným řešením aplikování detekčních metod na strojový kód.

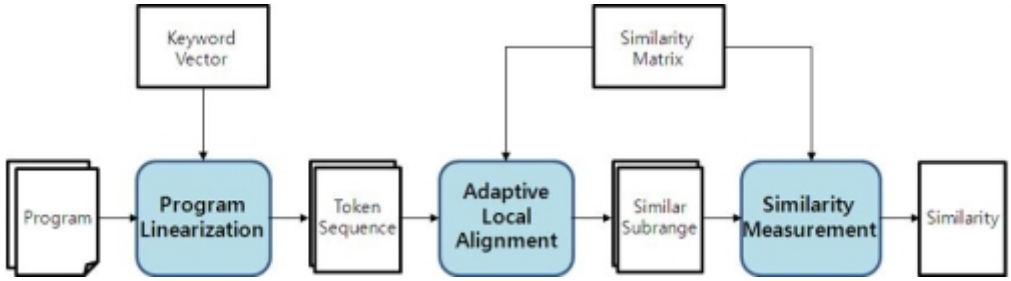
4.1 Přehled

Problémovou sférou plagiátorství je plagiování zdrojových kódů mezi studenty. Průzkum z roku 2006 mezi několika univerzitami z Velké Británie se zabýval pohledem profesorů a vyučujících na plagiátorství zdrojových kódů mezi studenty. V studii bylo přiblíženo, co lze označit za plagiátorství, většinou se jednalo o „zkopírování “, použití a neuvedení zdroje, znovupoužití vlastního kódu z jiného projektu, ač to bylo vysloveně zakázáno, ale také případy, kdy dva nezávisle vytvořené programy mohou být podobné a tudíž vyhodnoceny jako plagiované. V některých případech je dokonce vyžadováno vytvoření kódu, který se systémům pro detekci jeví jako plagiovaný, například když je skupině studentů zadáno naprogramování konkrétního algoritmu nebo se používají některé principy OOP jako refaktorizace, generování implicitního GUI kódu nebo jiné OO paradigmaty. Více informací je ve studii Cosmy a Joye [4].

V detekci plagiátů neexistuje referenční model, na jehož základě bychom mohli tvrdit, že první ze dvou podobných programů plagiarizoval druhý nebo naopak (pokud neporovnáváme program s existující databází starších programů). Pokud jsou dva programy podobné nebo napsány pro stejné zadání, potřebujeme pravděpodobnostní model pro takové případy. Dále by měl být detekční systém kvantitativní a velmi specifický v plagiátorských technikách a oblastech, kde se používá [1]. Takový systém, by měl být schopen určit, ve kterých oblastech došlo k plagiátorství a také jakým způsobem byl kód změněn.

Detekční systémy pracující na sémantické úrovni jsou velmi nákladné na výpočetní jednotky, navíc je vždy potřeba manuálního posouzení. Nabízí se tedy, stejně jako v jiných přístupech, pokrytí detekce na syntaktické úrovni, rozšíření i na plagiáty strukturální úrovně. Tato metoda detekce plagiátů ve zdrojových kódech se zaměřuje na vyhledávání bloků plagiovaného kódu, používající variantu lokálního zarovnání se zařazením pozměněné matice podobnosti tak, aby odrážela frekvenci klíčových slov skupiny zkoumaných programů, kdy každý prvek matice podobnosti je logaritmem pravděpodobnosti frekvence výskytu klíčového slova mezi skupinou testovaných programů.

Toto je model navržený Limem, Wooem a kol. [1] pro adaptivní porovnání podobnosti dvou programů. ALA používá lokální zarovnání a pracuje ve třech fázích a to: linearizace programu, lokální zarovnání a vyhodnocení podobnosti, graficky znázorněných na obr. 9.



Obrázek 9: Vyhodnocení podobnosti s použitím adaptivního lokálního zarovnání [1].

Dále je možné použít tento model pro vyhledání podobnostní mezi strojovými kódy. Tento postup byl aplikován autorem Ji a kol. [6] na bytekód. Narozdíl od jiných nástrojů, které provádí detekci plagiátů nebo klonů, přináší tento postup další úroveň, na které lze provádět detekci.

4.2 Linearizace programu

Jak bylo řečeno, prvním krokem je linearizace programu. Ta se skládá ze dvou částí. V první části probíhá tokenizace, která je založena na principu popsaném v 3.3.2. Nyní se vytváří vektor klíčových slov, tvořící množinu tokenů, které sloužily k tokenizaci, a to znamená, že je plně závislý na programovacím jazyku, použitém pro vytváření testovaných programů. Klíčová slova také zastupují operátory, separátory a další logické prvky, aby byly brány v potaz i strukturální znaky programu, kontrolní tok, podprogamy, bloky programu atd.

Druhá část linearizace obsahuje proces nazvaný podle autorů [1] statickým trasováním. To je technika, která spouští program na syntaktické úrovni a vygeneruje tedy posloupnost tokenů v pořadí, v jakém se spouští při chodu aplikace. Vytváří tak syntaktický strom,

jenž pak trasuje vykonávání příkazů, potažmo tokenů. Na obr. 10 je vidět, jak linearizace probíhá.

Source Code	Token Sequence

1 int main()
2 {
3 int num1 = 100;
4 int num2 = 200;
5 swap(&num1, &num2);
6 printf(num1 = %d,
num2 = %d", num1, num2);
7 }
8 void swap(int *m, int *n)
9 {
10 int *temp;
11 temp = m;
12 m = n;
13 n = temp;
14 }
	1 FUNC_CALL, # main()
	2 BLOCK_START
	3 INT, ASSIGNMENT
	4 INT, ASSIGNMENT
	5 REFERENCE, REFERENCE
	8 FUNC_CALL, # swap(int*, int*)
	9 BLOCK_START
	10 INT, PTR
	11 ASSIGNMENT
	12 ASSIGNMENT
	13 ASSIGNMENT
	14 BLOCK_END, # swap
	6 UNREACHABLE_FUNC, # printf
	7 BLOCK_END, # main

Obrázek 10: Ukázka linearizace programu [1].

Jedná se o příklad záměny dvou hodnot celočíselných proměnných pomocí funkce `swap`. Tokenizace se statickým trasováním generuje sekvenci tokenů a je znázorněna na obr. 10 vpravo, kde řádkování odpovídá vykonávání příkazů. Když linearizace narazí na volání funkce `swap` a ta je definována uživatelem, pak je volání metody nahrazeno její implementací (Volání metody `FUNC_CALL` a příkazy v jejím těle: řádek 8 - 14). Pak pokračuje dalšími příkazy z metody `main`. Pokud narazí na systémovou funkci, kterou nelze trasovat (nalézt) v kódu, přiřadí pouze token `UNREACHABLE_FUNC`.

Musím poznamenat, že v implementaci a návrhu algoritmu používám vlastní přístup při určování zrnitosti porovnávaných jednotek kódu (v rámci celého programu, souborů, metod nebo bloků). Popis vlastního přístupu je rozebrán v kap. 6.

4.2.1 Linearizace bytekódu

V případě linearizace bytekódu je princip generování sekvence tokenů principiálně stejný linearizaci zdrojového kódu, ale kvůli struktuře java `class` souboru se proces odlišuje.

Proces detekce plagiátů na úrovni bytekódu sestává z několika částí, jak navrhli Ji a kol. [6], ale v zásadě je podobný s postupem v detekčním systému pro zdrojové kódy navrženým podle Limem a kol. [1], na kterém se podíleli i autoři prvního přístupu. Liší se pouze v části linearizace programu a typem zpracovávaných vstupních dat. Fáze detekce podobnosti používá stejný algoritmus adaptivního lokálního zarovnání a také stejné

výpočty pro měření podobnosti, a proto si popíšeme odděleně pouze proces zpracování vstupních dat.

Z prvního pohledu se aplikují dva procesy pro vyhledávání plagiátů s použitím bytekódu, a to vytvoření sekvence tokenů a ohodnocení podobnosti dvou vstupních entit - programů. Jak jsem uvedl výše, druhá fáze je totožná se zpracováním zdrojového kódu podle [1].

Zpracování java class souboru

Vytvoření linearizované sekvence tokenů bytekódu probíhá načtením hlavní metody souboru (metoda `main` v java class souboru). Proces pokračuje načtením všech tříd náležícím java programu, díky zpracování jedné z částí class souboru - constant poolu. Ta obsahuje informace pro linkování všech tříd, které jsou potřeba pro běh programu. Sekvence tokenů bytekódu je vytvořena statickou analýzou metod, to je je proces, který staticky spouští bytekód instrukce na úrovni metod a extrahuje tokeny podle bytekód operace z každé metody podle pořadí jejího vykonávání [6].

Struktura java class souboru

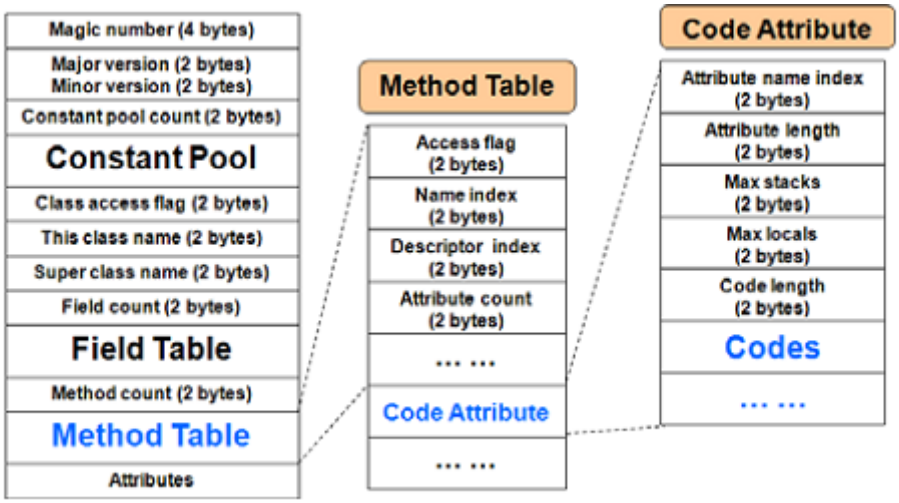
Bytekód se skládá z množiny binárních instrukcí, obecně se operace bytekódu skládá z opkódu a jeho operandu, takovou instrukci spustí JVM. Java bytekód zahrnuje 203 typů instrukcí, které produkuje kompiler do class souboru. Celý java program se pak skládá z množiny takových souborů. Soubor typu class obsahuje hlavičku souboru, constant pool, tabulku polí, metod a atributů atd. Constant pool udržuje informace o všech konstantách souboru, tabulky metod a polí udržují informace o umístění metod a polí, stejně tak tabulka atributů. Tyto informace se používá JVM pro debuggování java programu, ukázka na obr. 11. Bytekód se nachází v atributech kódu v tabulkách metod. Pro více podrobností doporučuju Java Virtual Machine Specification [2].

Nejdůležitější jsou kódové atributy v tabulkách metod, které obsahují bytekód, který se využívá v detekci plagiátů. Samotná linearizace bytekódu je tedy proces vytvoření sekvence tokenů analýzou bytekódu metod z class souborů. Ten probíhá ve dvou fázích, první je shlukování bytekód instrukcí a druhou je trasování v grafu volání metod.

Proces linearizace bytekódu

Shlukování bytekódu spočívá ve vytvoření kategorií, které zastupují podobné bytekód instrukce, vzhledem k jejich funkci. Například se jedná o instrukce `dstore`, `dstore_0`, `dstore_1`, atd., které ukládají reálnou hodnotu do lokální tabulky proměnných. Liší se pouze v lokaci, kde ukládají hodnotu ze zásobníku. V procesu shlukování jsou instrukce,

[2] Lindholm, Tim, Yellin, Frank, Bracha, Gilad, Buckley, Alex. The Java® Virtual Machine Specification, Java SE 7 Edition. 2013. Dostupné z: http://docs.oracle.com/javase/specs/jvms/se7/html/



Obrázek 11: Grafická podoba struktury Java class souboru [6].

které mají podobnou funkci seskupeny do skupiny a z 203 instrukcí neboli opkódů vytvořili autoři 59 skupin, druhů tokenů [6]. Díky této redukci počtu srovnávaných prvků dosahuje systém zvýšení výkonu detekce, jelikož některé operace lze zaměnit bez podstatného změnění výstupu. Záměna pořadí proměnných, jako jedna z metod modifikace kódu, je takto odstraněna.

Druhá fáze odráží charakteristiku programového toku, používá metodu trasování v grafu volání metod. Zde je použit orientovaný graf, kde jednotlivé uzly představují metody a hrany jednotlivá volání dalších metod, ukazatele, která metoda používá ve svém těle další, případně tvoří cyklus a jedná se o rekurzivní volání metod. Po sestavení grafu můžeme vyřadit nedosažitelné funkce, jelikož k nim nevede žádná hrana, takže se zbavíme další plagiátorské modifikace a nakonec i záměny sekvence příkazů za volání metody. Na výstupu linearizace je tedy sekvence tokenů, navíc seřazených podle pořadí, ve kterém jsou metody volány během spuštění programu.

4.3 Adaptivní lokální zarovnání

Jak již bylo řečeno, adaptivní lokální zarovnání je variací Smith - Watermanova algoritmu lokálního zarovnání (více v 3.4.2) s ohledem na matici podobnosti. Podobnost dvou programů se počítá s přihlédnutím na celou množinu programů, které zahrnují testovanou dvojici. Zde se odráží srovnávací skóre („matching score“) klíčových slov, které se počítá podle jejich frekvence výskytu ve skupině programů, viz. tab. 3. Aby neměla určitá běžnější slova větší váhu než ty raritnější, počítá se srovnávací skóre reverzně, tedy klíčová slova s nízkou frekvencí výskytu mají vysoké skóre a slova s vysokou frekvencí mají nízké skóre.

Klíčové slovo	Frekvence	Klíčové slovo	Frekvence
(,)	9,704 %	switch	0,064 %
}	4,338 %	protected	0,096 %
{	4,338 %	default	0,032 %
=	4,21 %	!	0,064 %

Tabulka 3: Klíčová slova s vysokou a nízkou frekvencí (vyjádřeno procentuálně).

4.3.1 Vektor klíčových slov

Pro výpočet matice podobnosti musíme zjistit frekvence výskytu klíčových slov. Celá skupina programů se skládá z n jednotlivých programů, tedy množina programů: $P = \{p_1, p_2, ..., p_n\}$ a počet výskytů klíčového slova k v programu p je označen jako $occur(p, k)$, pak se celkový počet výskytů klíčového slova k v množině programů P definuje jako:

Definice 4.1

$$occur(P, k) = \sum_{p \in P} occur(p, k)$$

Za základě těchto vztahů definujeme frekvenci f klíčového slova k_i v množině programů P takto:

Definice 4.2

$$f_i^P = \frac{occur(P, k_i)}{\sum_{j=1}^{r} occur(P, k_j)}$$

Poznámka 4.1 Jelikož jmenovatel $\sum_{j=1}^{r} occur(P, k_j)$ označuje součet počtu výskytů všech klíčových slov, frekvence f_i^P klíčového slova k_i náleží do intervalu $(0 \leq f_i^P \leq 1)$.

4.3.2 Matice podobnosti

Podobně jako v případě lokálního zarovnání, kde se používá skórovací schéma, používáme matici podobnosti, tedy jiný druh schématu. Jedná se o matici SM, která má rozměr $(r + 1)x(r + 1)$, kde r je počet druhů klíčových slov a $+1$ je místo pro prázdný symbol („gap“ nebo také mezeru) na $(r+1)$-tém řádku a sloupci matice podobnosti. Každý prvek matice $SM(k_i, k_j)$ představuje skóre nebo postih (ohodnocení) za shodou dvou klíčových slov nebo jejich neshodu, jsou-li různé. Shoda nastává v případě, že $k_i = k_j$ a neshoda, pokud $k_i \neq k_j$. Pro řádky a sloupce s prázdným symbolem představují prvky matice postihy za vkládání nebo mazání prázdných symbolů.

Použitím frekvencí klíčových slov můžeme nyní přejít k definování adaptivní skórovací matice podobnosti SM^P takto:

$$SM^P(k_i, k_j) = \begin{cases} -\alpha \cdot log_2(f_i^P \cdot f_j^P) & if\ k_i = k_j - Match \\ \beta \cdot log_2(f_i^P \cdot f_j^P) & i\ f k_i \neq k_j - Mismatch \\ 4\beta \cdot log_2 \cdot f_i^P & if\ k_j\ is\ gap\ and\ k_i\ is\ not - Deletion \\ 4\beta \cdot log_2 \cdot f_j^P & if\ k_i\ is\ gap\ and\ k_j\ is\ not - Insertion \\ -\infty & if\ k_i\ and\ k_j\ are\ both\ gaps \end{cases}$$

Ladící parametry α a β jsou nastaveny na součet 1 ($\alpha + \beta = 1$), můžeme jimi měnit relativní váhy pro skóre shody a postih za neshodu. Nastavením $\alpha > \beta$ získáme vyšší ohodnocení pro skóre shody a tedy i v celkové míře podobnost programu a reverzně pro $\alpha < \beta$. O experimentálním testování těchto parametrů se můžete dočíst v [1].

Prvky matice podobnosti jsou vytvořeny na základě binárního logaritmu frekvence klíčových slov. Porovnáváme vždy dva prvky, tedy vezmeme v potaz tzv. „odds“ neboli šance, která udává poměr mezi pravděpodobností výskytu jevu k pravděpodobnosti nevýskytu kombinace nebo klíčového slova. Pak vztah mezi těmito hodnotami určuje logaritmus šance „log odds“, který se např. používá pro porovnání intenzity dvou signálů s velkým rozdílem poměru intenzity [1]. Tento rozptyl hodnot vidíme u tab. 3.

Pomineme-li ladící parametry, tak se skóre shody neboli ohodnocení shody skládá z logaritmického vztahu produktu frekvencí dané skupiny klíčových slov. Podobně to platí pro ohodnocení neshody - postih, akorát se záporným ohodnocením. Ohodnocení vkládání nebo mazání mezer je také postihováno záporným ohodnocením, ale 2-krát větším, jak vidíme v skórovacím schématu. V Případě vkládání dvou prázdných symbolů je situace taková, že vložíme záporné nekonečno, ikdyž by tato situace neměla nastat [1].

Na základě výše zmíněných definicí a popisu Smith - Watermanova algoritmu, z kap. 3.4.2, jsem definoval výpočet adaptivní matice podobnosti se skórovací maticí $SM^{(P,Q)}$, odrážející frekvenci klíčových slov mezi dvěma programy p_m a q_n, kde p je m-tý program z množiny programů P, a kde q je n-tý program z množiny programů Q, a s vektorem klíčových slov výpočítaným pro tyto množiny programů jako $H(p_m, q_n)$:

$$H_{p_m,q_n}(i,j) = \begin{cases} H[i-1][j-1] + SM[i][i] & if\ i = k - Match \\ H[i-1][j-1] + SM[i][j] & i\ f k_i \neq k_j - Mismatch \\ H[i-1][j] + SM[i][-] & if\ j\ is\ gap\ and\ i\ is\ not - Deletion \\ H[i][j-1] + SM[-][j] & if\ i\ is\ gap\ and\ j\ is\ not - Insertion \end{cases}$$

4.4 Měření podobnosti

Hodnotu míry podobnosti pro zarovnané oblasti mezi dvěma programy spočítáme pomocí adaptivní matice podobnosti. Samotné zarovnání probíhá během počítání míry podobnosti zarovnání, avšak pro nás je vhodné konceptuálně rozdělit tyto dva procesy

do pořadí, kdy zarovnání probíhá před samotným počítáním míry podobnosti dvou programů [1].

Funkce zarovnání je označena jako *align* a jejím výstupem je dvojice zarovnaných bloků srovnávaných souborů programů A a B. Jednotlivé bloky zarovnání z A, $\langle a_1, a_2, ..., a_m \rangle$ a k nim korespondující zarovnání B, $\langle b_1, b_2, ..., b_m \rangle$, výstupem funkce *align* je vektor korespondujících dvojic klíčových slov:

$$align(A, B) = \langle (a_1, b_1), (a_2, b_2), ..., (a_m, b_m) \rangle$$

Nyní definujme absolutní míru podobnosti dvou programů s použitím funkce *align* takto:

Definice 4.3

$$SIM_{abs}(A, B) = \sum_{(a,b) \in align(A,B)} H^P(a, b)$$

Absolutní míra podobnosti dvou programů je sumací jednotlivých hodnot podobnosti korespondujících klíčových slov a mezer v zarovnaném bloku.

Abychom mohli porovnat dvě různé míry podobnosti dvou programů, potřebujeme určit normalizovanou míru podobnosti. Jelikož SIM_{abs} závisí na délce srovnávaných programů, nemůžeme objektivně srovnávat podobnost dvou programů pomocí tohoto parametru.

Způsob, jak normalizovat absolutní míru podobnosti, je určení podílu míry podobnosti součtem měr podobnosti. Dostaneme tedy podle autorů [1] funkci pro určení míry podobnosti $SIM_{sum}(A, B)$ takto:

Definice 4.4

$$SIM_{sum}(A, B) = \frac{2SIM_{abs}(A, B)}{SIM_{abs}(A, A) + SIM_{abs}(B, B)}$$

Poznámka 4.2 Pokud jsou A a B identické, je absolutní míra podobnosti vynásobena 2, aby normalizovaná míra podobnosti byla ve výsledku rovna 1 (tedy oba porovnávané programy jsou zcela totožné).

V případech, kdy jsou plagiarizovány pouze některé části programu a délka původního programu je značně větší než plagiátu, mohou rozdíly v délkách programů způsobit, že míra podobnosti bude menší, a tak bude plagiarizovaný program ohodnocen malou mírou podobnosti a může být vyloučen z detekce. Je-li plagiát kompletně sestavený z jiného programu, měla by být míra podobnosti 100 %, ale SIM_{sum} není [1]. Podobně můžeme nahlížet na problém vkládání velkého množství nesmyslných příkazů a nedosažitelných stavů. Autoři [1] navrhují následující počítání míry podobnosti:

Definice 4.5

$$SIM_{min}(A, B) = \frac{SIM_{abs}(A, B)}{min\{SIM_{abs}(A, A), SIM_{abs}(B, B)\}}$$

V závislosti na povaze modifikace kódu můžeme použít některé funkce nebo kombinace funkcí měření míry podobnosti.

Míra podobnosti dvou kolekcí tokenů

Tento parametr slouží k zjištění procentuální podobnosti výskytů stejných tokenů nebo prvků mezi dvěma kolekcemi. Využívám tento výpočet při detekci podobností konfigurací zdrojových kódů a vztahují se na importované knihovny porovnávaných programů.

Jestliže máme dvě kolekce importů, kde každá je získaná z množiny zdrojových kódů pro daný program, a žádný import se nevyskytuje v rámci svého programu dvakrát, tedy každá kolekce obsahuje pouze jedinečné položky, definujeme jejich podobnost pomocí parametrů: l_1 - počet jedinečných importů v první kolekci, l_2 jako počet jedinečných prvků v druhé kolekci, Q je počet stených prvků mezi dvěma kolekcemi a U jako součet všech unikátních importů obou kolekcí s výpočtem:

$$U = Q + ((l_1 - Q) + (l_2 - Q))$$

Pak je procentuální míra podobnosti SIM_I, mezi těmito dvěma kolekcemi dána takto:

$$SIM_I = \frac{Q \cdot 100}{U}$$

4.5 Závěr

Na základě experimentálních testů byla otestována tato metoda a distribuce pravděpodobnosti odhalení plagiátů byla větší a odhalení citlivější, než výsledky lokálního zarovnání s použitím fixní matice podobnosti (ohodnocující schéma) s +2 body za shodu, -1 za neshodu a -2 pro srovnání s mezerou. Výkon ALA je nadřazený výkonu GST, jakožto dvou podobných metodologií [1].

Jelikož se při linearizace programu vytváří syntaktický strom, který určuje pořadí spouštění jednotlivých metod programu, vytváří se sekvence tokenů pro jediný, sloučený soubor a lokální zarovnání probíhá pro celou takovou sekvenci. Ve vlastní práci upřednostňuji jednodušší návrh, který nepoužívá statické trasování kvůli nutnosti zpracování zdrojového kódu na syntaktické úrovni, což by směřovalo k syntaktické analýze programu. Navíc porovnávám programy na úrovni souborů (metod), abych dosáhl přesnějších výsledků lokálního zarovnání Smith - Watermanova algoritmu v kombinaci s adaptivní maticí podobnosti.

Ve své práci, a hlavně pak při návrhu systému pro detekci plagiátů, jsem bral ohled na již existující řešení v této oblasti. Velmi častým přístupem pro detekci plagiátů mezi zdrojovými kódy je porovnávání řetězců. Nicméně, tyto algoritmy mají malou účinnost při modifikacích vyšších úrovní a plagiarizované programy mohou uniknout odhalení. Na

druhou stranu přístupy, které vytvářejí v procesu transformace složitější struktury, jako například AST nebo PDG, jsou výpočetně náročné, ale odhalí vysokoúrovňové modifikace. ALA algoritmus zahrnuje výpočetně nenáročnou lexikální analýzu (bez linearizace) pro intermediální reprezentaci kódu a detekce probíhá na základě lokálního zarovnání se skórovacím systémem, který reflektuje porovnávané entity v rámci množiny programů, kde testování probíhá. Tedy, jistou měrou zahrnuje i syntaktickou strukturu zpracovávaného kódu, a je tudíž ideálním kandidátem v detekčním procesu.

5 Návrh aplikace pro detekci plagiátů

V této kapitole je popsán návrh a požadavky aplikace, která řeší problematiku vyhledávání plagiátů ve zdrojových kódech. Snažil jsem se o vytvoření aplikace, která by splňovala kritéria zadání diplomové práce, tedy porovnávala míru podobnosti dvou vstupních programů napsaných v programovacím jazyce Java, a to navíc na několika úrovních. Jako ve většině existujících detekčních systémů a řešení probíhá vyhledávání na úrovni nekompilovaných zdrojových kódů, v mém případě se jedná o soubory formátu `java`. Další úrovní, kterou integruji ve svém systému, je úroveň podobnosti bytekódu, kdy jsou vstupem java class soubory. Dále probíhá porovnání mezi soubory ve formátu dekompilovaných java souborů. A nakonec se zaměřuji na podobnost konfigurace zdrojových kódu, kdy je podobnost hledána v použitých knihovnách, převážně se jedná o importy z knihoven Java API.

5.1 Požadavky aplikace

Aplikace srovnává podobnost dvou vstupních programů napsaných v Javě. Vstupními parametry příkazové řádky při spuštění aplikace jsou dvě cesty k adresářům, které obsahují Java projekty, jeden považován aplikací za originální a druhý za podezřelý. Aplikace vytřídí relevantní soubory, které potřebuje pro detekci, jedná se o soubory formátu `java` a `class`. Soubory jsou předzpracovány a je provedena některá z forem normalizace (odstranění formátování, whitespace znaků, komentářů apod.). Zdrojový kód v souborech je transformován do intermediálních interních struktur, které jsou vhodnější pro detekční algoritmy a odstraňují některé formy plagiátorských modifikací. Následuje detekce shod, vyhodnocení míry podobnosti a mapování na původní, originální soubory se zdrojovým kódem tak, aby byla zajištěna nejvyšší možná úroveň odhalení plagiátů. Výsledky celého procesu detekce jsou agregovány do HTML souboru, který označuje procentuální míru podobnosti mezi programy a zobrazuje plagiované pasáže kódu. Rozlišení, který program plagioval od kterého, je na posuzovateli.

5.2 Přehled

Hlavní inspiraci jsem čerpal v práci autorů [1] a [6], kteří využívají lexikálně-syntaktickou analýzu zdrojových kódů, jako intermediální reprezentaci kódu sekvenci tokenů a vyhledávání podobnosti na základě lokálního zarovnání s maticí podobnosti odrážející frekvenci klíčových slov, které se vyskytují v porovnávaných programech.

Vlastním přínosem v této práci je integrace výše zmíněných řešení a postupů do uceleného systému, zkoumání a navržení řešení nových, které lze vidět převážně v zařazení dekompilovaných kódů a metodologie vyhledávání podobností na úrovni mezikódu v procesu detekce, a také vlastní přístup při implementaci problému.

5.3 Návrh aplikace

Při vývoji aplikace jsem bral zřetel na její funkcionalitu a výsledkem je systém, který se drží popsaného procesu detekce plagiátů z kap. 3.2. První částí je proces zpracování vstupních souborů, jejich dekompilace, případně „disasemblování" do textového formátu a vytvoření adresářové struktury v dočasném adresáři `Temp`, defaultně umístěným v kořenovém adresáři aplikace. Zároveň jsou tyto vstupní soubory neboli testovací entity načteny do kolekcí a postoupeny k další fázi zpracování v rámci PDA aplikace (Plagiarism Detection Application).

Následuje proces lexikální analýzy, kdy je každý soubor kolekce analyzován vygenerovaným skenerem z programu JFlex a všechny soubory jsou takto tokenizovány na základě slovníků pro daný programovací jazyk, v tomto případě pro Java kód a Java bytekód. Takže nynější podoba interního zpracování dat je sekvence tokenů, reprezentující každý vstupní soubor. Dalším procesem je druhé kolo tokenizace, kdy jsou některé struktury a podobné skupiny tokenů převedeny na obecnější zástupce, díky čemuž dojde k zvýšení odolnosti aplikace proti některým plagiátorským modifikacím, obdobně postupoval ve své práci i autor [17].

V rámci programových jednotek, které v aplikaci reprezentují všechny typy souborových jednotek pro vstupní projekty, u kterých hledáme podobnost, počítáme vektory klíčových slov pro každou skupinu porovnávaných souborů, takže jsou vytvořeny vektory pro java soubory, bytekód soubory a dekompilované soubory. Z nich se počítají matice podobnosti reflektující frekvence klíčových slov pro danou množinu programů.

Následuje proces samotné detekce podobnosti založený na Smith - Watermanově algoritmu. Je vytvořena adaptivní matice podobnosti pro dvě sekvence tokenů a je vypočítáno nejoptimálnější lokální zarovnání pomocí algoritmu zpětného trasování.

Posledními kroky jsou vypočítání míry podobnosti pro dané páry srovnávaných souborových jednotek a vytvoření reportu ve formě HTML souboru, obsahující konkrétní procentuální míy podobnosti pro páry srovnávaných jednotek daných programů a případně zobrazení původního kódu, kde byla detekována podobnosti. V tomto kroku probíhá také zpětné mapování odhalených podobností na původní vstupní soubory.

Obecně lze popsat proces PDA systému v posloupnosti kroků:

- *Inicializace aplikace (Application inicialization)* - Vytvoření adresářové struktury, načtení parametrů příkazové řádky, tedy načtení adresářů s porovnávanými projekty.
- *Předzpracování souborů (File preprocessing)* - Filtrování požadovaných souborů podle jejich formátu, kopírování do kořenového adresáře aplikace, disasemblování class souborů, dekompilace a nahrání do kolekce souborových jednotek pro další zpracování.

- *Transformace (Transformation)* - Fáze transformace zahrnuje lexikální analýzu, převedení textových řetězců na sekvenci tokenů a druhé kolo tokenizace pro zobecnění některých původních Java literálů. Zároveň se provádí preprocessing, jak je vysvětlen v kapitole 3.2.1, jelikož při lexikální analýze dochází k odstranění komentářů, formátování, odstranění bílých mezer a zobecnění názvů proměnných.

- *Výpočet vektoru klíčových slov a matice podobnosti (Keyword vector and similarity matrix computing)* - Tato fáze probíhá zároveň s dalším bodem, ale můžeme je konceptuálně oddělit od sebe. Načítají se všechny tokeny do vektorů klíčových slov v závislosti na typu srovnávaných souborů. Z nich se vytvoří matice podobnosti reflektující frekvenci tokenů.

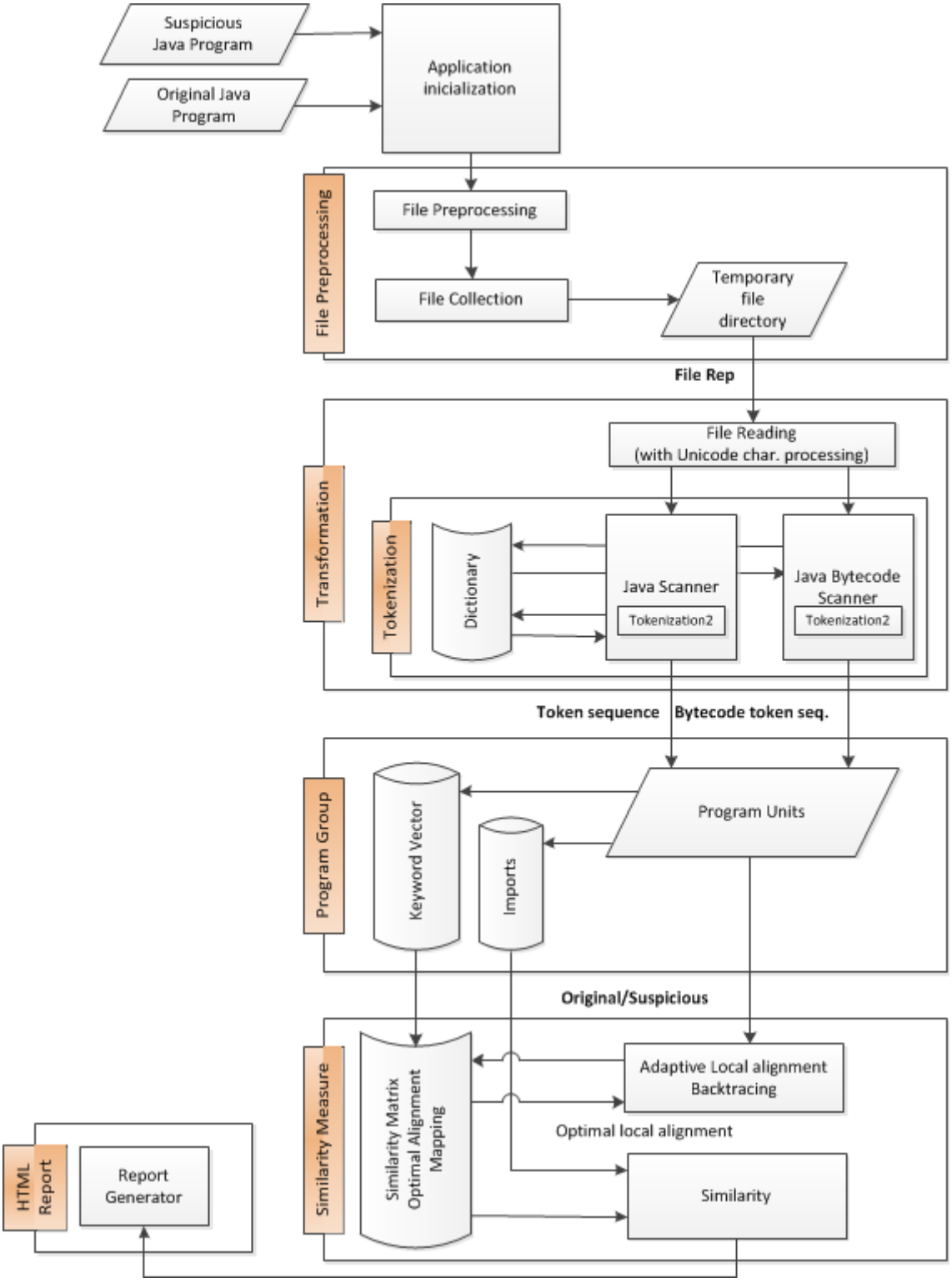
- *Detekce shody (Match detection)* - Využitím lokálního zarovnání pomocí Smith - Watermanova algoritmu se vypočítá adaptivní matice podobnosti pro dva řetězce tokenů a aplikací zpětného trasování se vypočítá optimální zarovnání mezi dvěma oblastmi. Vytváří se také body, které označují začátky a konce zarovnání, reflektující fragmenty kódu z původních souborů, kde byla nalezena vysoká míra podobnosti.

- *Výpočet míry podobnosti (Similarity measurement)* - Podle vzorců uvedených v 4.4 se vypočítává míra podobnosti pro různé dvojice souborů podle jejich kategorie: pro java soubory, bytekód soubory, dekompilované soubory a mezi java a dekompilovanými soubory.

- *Mapování detekovaných párů s vysokou podobností (High similarity pair mapping)* - Zde jsou namapovány všechny páry s vysokou mírou podobnosti na základě optimálního lokálního zarovnání na původní fragmenty kódu v originálních souborech.

- *Vizualizace výsledků (Visualization)* - Statistiky procentuální podobnosti a optimálních lokálních zarovnání jsou vygenerovány formou HTML reportu.

Návrh architektury aplikace je na obr. 12 a detailní popis jednotlivých kroků procesu detekce systému je rozepsán v následující kapitole 6.



Obrázek 12: Návrh architektury aplikace.

6 Implementace aplikace

Tato kapitola popisuje aplikační algoritmy systému pro detekci plagiátů v Java programech. Jednotlivé podkapitoly popisují jednotlivé algoritmy, jejich strukturu a případně jakým problémům jsem čelil při jejich implementaci.

Události aplikace, výjimky a chyby v běhu programu jsou zaznamenávány logovací knihovnou Log4j do dvou souborů, které se nachází v kořenovém adresáři aplikace `Log`. Nastavení chování loggerů se provádí přes `log4j.properties` soubor, podrobnosti lze najít na stránkách Log4j projektu [30]. Soubor `info.log` zaznamenává spouštění procesů v aplikaci, tedy lze z něj vyčíst běh celé aplikace a `error.log` soubor, kde jsou zaznamenávány výjimky a chyby programu. Inicializace logování se provádí v hlavní třídě programu: `PropertyConfigurator.configure("log4j.properties")`, kde se nastavuje konfigurace. Volání událostí a jejich zapisování do patřičných souborů se provádí definicí privátní konstanty v každé třídě, kde chceme logger použít:

```
private static final Logger logger = Logger.getLogger(PDApplication.class);
```

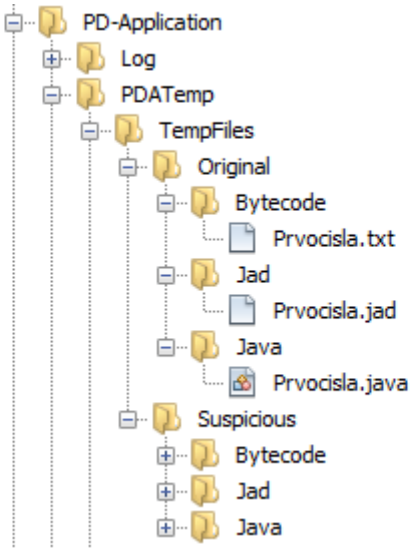
Samotný zápis logovací zprávy probíhá přes metody proměnné `logger`: `info("msg")` nebo `error("msg")` a zapisuje informace do souborů.

6.1 Inicializace aplikace

Algoritmus prvního procesu aplikace by měl, podle požadavků, načíst při spuštění aplikace také další dva parametry příkazové řádky, které označují cestu k adresářům, které obsahují programy určené k porovnání PDA systémem. Vstupní parametry jsou načteny do řetězců `originalPath` a `suspiciousPath` a následuje ověření existence adresářů v těchto cestách.

Pokud nebyly zadány vstupní parametry, aplikace načte soubory pro detekci z testovacích projektů nacházejících se na přiloženém médiu. V případě úspěšného vložení vstupních parametrů budeme dále se soubory se zdrojovými kódy pracovat, a proto musíme vytvořit adresářovou strukturu v kořenovém adresáři aplikace, kde budeme ukládat jednotlivé typy souborů, náhled na obr. 13. Při implementaci jsem využil standardní metody třídy `java.io.File` a příkaz operačního systému Windows spuštěný přes třídu `Proces` prostředí Javy pro vykonávání nativních procesů. Nechybí zachytávání výjimek do logovacího souboru a ověřování vytvoření objektů. Jelikož se jedná o triviální problém, není třeba jej dále rozepisovat.

Dále musíme systému zadat některé parametry, důležité pro ladění algoritmu adaptivního lokálního zarovnání a filtrování výsledků s minimální podobností. Zde byla použita třída `Parameters` s implementací Java třídy `Properties` pro vytvoření persistentní množiny klíčů a hodnot ve formě souboru `config.properties`. Tento soubor obsahuje cesty k porovnávaným projektům, limit (`treshold`) pro vyřazení nalezených párů



Obrázek 13: Adresářová struktura PDA aplikace

s malou podobností a ladicí konstanty `alpha` a `beta`, jejichž význam je vysvětlen v kap. 4.3.2.

6.2 Předzpracování souborů

Předzpracování souborů je první fáze v procesu detekce. Jak je vidět na obr. 12, skládá se ze dvou kroků a jeho výstupem je dočasné lokální uložiště pro zpracovávané soubory. Prvním krokem je `FilePreprocessing`, který obstarává vytvoření požadovaných formátů a `FileCollection`, který reprezentuje jednotlivé kolekce pro všechny typy zpracovávaných formátů souborů.

V první řadě je inicializován objekt třídy `FileCollection` pro každý ze vstupních parametrů příkazové řádky, tedy systémová cesta pro originální nebo podezřelý program. V objektu jsou vytvořeny relativní cesty k dočasnému uložišti, kde se budou nahrávat jednotlivé již zpracované soubory. Dále je běh programu předán instanci třídy `FileProcessing` i s parametrem cesty k adresáři programu. Tato instance má definovány dva druhy kolekcí, jeden pro soubory formátu `java` a druhý pro formát `class`.

Po inicializaci jednotlivých `ArrayList` struktur je zavolána metoda `getCollection`, která načte všechny soubory do kolekce, odstraní adresáře a podle masky vyfiltruje soubory potřebného formátu. Jedná-li se o formát java, je zavolána metoda pro kopírování, která používá metodu finální třídy `RuntimeCommand`, `copyJavaFile` a zkopíruje všechny java soubory do dočasného adresáře podle příslušného programu, viz. obr. 13. Pokud se jedná o soubory typu `class`, tak jsou načteny stejně jako `java`, ale poté se provedou dva příkazy třídy `RuntimeCommand`. První disasembluje java class soubory

metodou executeJavap, která spouští java příkaz javap -c -private pro každý soubor a jeho textovou podobu předává na standardní výstup, tedy je ještě nutné jej přečíst BufferedReader proudem a zapsat jej do souboru FileWriter výstupním streamem. Vstupní parametr metody boolean original určuje jestli se soubory ukládají do adresářů pro originální nebo podezřelé soubory. Tento proces celkově zpomaluje běh programu, jelikož buffer musí čekat, než se vytvoří standardní výstup.

Pomocí další metody RuntimeCommand.executeJad jsou class soubory dekompilovány příkazem jad.exe nástroje JAD. Opět se vytvoří v příslušné složce dočasných adresářů. Takto končí zpracování souborů.

Nyní se načtou podle metody getCollections, která přijímá jako parametry masku formátu souboru a jeho cestu, soubory do objektů kolekcí (atributy objektu: kolekce FileRep - javaFiles, jadFiles a bytecodeFiles) instance reprezentativních tříd poděděnných z FileRep podle typu: JavaFile, pro bytekód soubor ClassFile a pro dekompilovaný soubor JavaDecompiledFile.

Jednotlivé atributy objektu FileCollection budou sloužit v dalším kroku jako vstupní data pro trasnformaci kódu na intermediální strukturu, v tomto případě sekvenci tokenů.

6.3 Transformace kódu

Tato fáze procesu detekce plagiátů transformuje vstupní programy, jejich soubory na sekvence tokenů. Takovou vnitřní strukturu reprezentace zdrojového kódu dokáže detekční algoritmus zpracovat mnohem efektivněji než například sekvence řetězců. Vyšší abstraktizace těchto struktur ignorují některé plagiátorské modifikace nižších úrovní. Proces transformace je rychlejší než při zpracování složitějších struktur jako parsovací stromy, AST nebo PDG, avšak méně odolný vůči modifikacím vyšších úrovní.

Datové struktury

Při transformaci pracuji s datovými strukturami, nad kterými probíhá detekce. Nejnižší jednotka, která reprezentuje Java literály, klíčová slova, atd., je třída Token. Třída FileUnit pak obsahuje informace o původním souboru, kolekci vytvořených tokenů a kolekci importů nalezených v původním souboru. Její obdobou je třída ByteFileUnit, která drží kolekci metod java class souboru a BytecodeMethod s kolekcí tokenů zastupujících jednotlivé opkódy a jejich parametry. Jsou zde popsány i vyšší struktury, se kterými se pracuje v dalších fázích procesu detekce.

Základní datové jednotky jsou:

- `Token` - je inicializovaný skenerem z lexikálního generátoru JFlex na základě specifikačního souboru pro jazyk Java 1.5. Je zobrazena v UML třídním diagramu na obr. 19
- `FileUnit` - třída, která drží sekvence tokenů a importů, je vyobrazena na obr. 19. Její obdobou je třída `ByteFileUnit`
- `ProgramUnit`- datová struktura, která seskupuje informace o jednotlivých souborech, obsahuje kolekce objektů `FileUnit` pro všechny typy porovnávaných souborů (java, bytekód, jad, importy)
- `ProgramGroup` - je reprezentativní struktura, která drží ve svém objektu vektory klíčových slov pro všechny srovnávané úrovně, programové jednotky pro originální a podezřelý program a importy z java souborů

6.3.1 Lexikální slovníky

Lexikální slovníky se používají při vytváření tokenů, kdy je hodnota řetězce přečtená skenerem nahrazena tokenem z datového slovníku s názvem konstanty a její hodnotou celého čísla (TOKEN_NAME = ID). Slovníky jsou implementovány formou rozhraní se seznamem konstant, např.: 2.

```
public interface IJavaDictionaryI {
  public static final int BYTE = 3;
  public static final int DOUBLE = 9;
  public static final int PRIVATE = 27;
  public static final int PROTECTED = 26;
  public static final int PUBLIC = 25;
  public static final int DO = 48;
  public static final int WHILE = 49;
  public static final int FOR = 50;
    ...
```

Výpis 2: Slovník pro tokenizaci implementovaný jako rozhraní

V programu se používá 5 slovníků podle jejich použití. Dva jsou určeny pro Java programovácí jazyk a tři pro skenování bytekódu. První slovník `IJavaDictionary` využívá skener pro vygenerování tokenů z java souboru a druhý slovník `IJavaDictionaryII` se využívá při druhé fázi tokenizace. Obdobná situace nastává při skenování bytekód souborů, slovník `IJavaBytecodeDictionary` slouží pro první kolo lexikální analýzy Java class souboru, další dva pak slouží při druhé fázi tokenizace k zobecnění některých skupin tokenů s podobným významem.

Generalizace v druhé fázi tokenizace znamená, že se určitá skupina tokenů převede na obecného zástupce této skupiny, například výše zobrazená část slovníku bude nahrazena tokeny ze slovníku `IJavaDictionaryII`, viz. výpis zdrojového kódu 3.

```
public interface IJavaDictionaryII {
  public static final int NUMBER = 200;
  public static final int ACCESS = 203;
  public static final int LOOP = 205;
    ...
```

Výpis 3: Slovník pro druhou tokenizaci

Tokeny `BYTE`, `DOUBLE`, atd. byly nahrazeny jedním obecným tokenem `NUMBER`. Skupina přístupových tokenů: `PRIVATE`, `PROTECTED` a `PUBLIC` je nahrazena jediným tokenem `ACCESS` a tokeny popisující klíčová slova cyklů `DO`, `WHILE` a `FOR` jsou nahrazena tokenem `LOOP`. Takto dochází při druhé tokenizaci k zobecňování některých skupin tokenů a důvodem je zvýšení efektivity algoritmu při hledání plagiátů, jelikož plagiátor může při modifikaci použít některou z plagiátorských změň, například změna datového typu, přístupového modifikátoru nebo změnou cyklu za jiný. Takto se aplikace stává odolnější vůči těmto modifikacím.

6.3.2 Lexikální analýza

Tokenizace neboli lexikální analýza spočívá v konverzi sekvence znaků na příslušný token. Tento proces provádí skener. Programovací jazyk Java obsahuje symboly, ze kterých se tvoří vyšší struktury jazyka. Skener tedy načítá sekvenci znaků, které mohou obsahovat symboly jako klíčová slova, separátory, operátory, literály a identifikátory. Princip je rozebírán v podkapitole 3.3.2. Skener pro aplikaci detekce plagiátů byl vytvořen z nástroje JFlex [21], obsahuje metody a pravidla pro zpracování všech Java symbolů, ze kterých vyprodukuje odpovídající tokeny s vlastnostmi: Celé číslo `line` pro pozici řádku, kde byl symbol nahrazen tokenem v původním souboru, řetězec `type` označuje řetězcovou hodnotu typu tokenu, získány díky vlasnosti Javy - reflexe ze slovníku pro první tokenizaci, celé číslo`id` označující typ tokenu a řetězec `value`, který reprezentuje původní sekvenci znaků, přečtených skenerem.

Java skener

Jelikož se při transformaci musíme zaměřit na dvě různé struktury souborů, ze kterých chceme vyextrahovat sekvence tokenů, musel jsem použít dva různé skenery s odlišnými slovníky pro tokenizaci. Prvním je java skener, reprezentovaný pro první fázi třídou `Scanner`, který implementuje rozhraní `IJavaDictionary`, ze kterého díky reflexi přiřazuje jednotlivým tokenům jejich typ a id. Ostatní vlastnosti objektu `Token` jsou získány z objektu třídy při skenování (původní hodnota, číslo řádku, atd.).

Metoda `getTokenName`, která přijímá jako vstup celé číslo, využívá díky slovníku znalost id čísla přečteného tokenu, ale musíme mu ještě přiřadit jeho typ pomocí reflexe polí Javy, kdy metoda vrátí řetězec názvu proměnné daného shodného id tokenu. Metoda `scanToken` pak vytváří objekt `Token` s danými daty, které jsou zpracovány během čtení sekvence znaků (`yyline`+1 parametr určuje číslo řádku původního řetězce, `s.sym` je

```
private String getTokenName(int token) {
    java.lang.reflect.Field [] classFields = IJavaDictionary.class.getFields();
    for (int i = 0; i < classFields.length; i++) {
      if (classFields[i].getInt(null) == token) {
        return classFields[i].getName();
     ...
}
public Token scanToken() throws java.io.IOException
{
  java_cup.runtime.Symbol s = next_token();
  Token token = new Token(yyline+1,s.sym,getTokenName(s.sym),yytext(),s.toString());
  return token;
}
```

Výpis 4: Metody scanToken a getTokenName třídy Scanner

id tokenu, `yytext` je přečtený řetězec, poslední atribut je hodnota řetězce, pokud byl přečten). Tato třída využívá knihovny CUP.

Třída, ktera používá objekt třídy `Scanner` a její operaci `scanToken` k vytvoření sekvence tokenů, je třída `JavaScanner`, která vygenerovanou sekvenci tokenů předává objektu `FileUnit`. Třída také implementuje rozhraní `IJavaTokenization`, popisující metody, které provádí druhou fázi tokenizace a je třeba je implementovat viz. 5. Metody

```
public interface IJavaTokenization {
    void createUnifiedIdentifier(FileUnit fileunit);
    Scanner fileReading(File file);
    boolean isAccessModifier(Token token);
    ...
    boolean isNumber(Token token);
    void level2Tokenization(FileUnit fileunit);
    void removeSemicolons(FileUnit fileunit);
    FileUnit scanner(FileRep file);
    void separateImports(FileUnit fileunit);
    Token tokenization(Token tkn);
```

Výpis 5: Rozhraní IJavaTokenization

jako `isNumber` ověřují na základě vstupního parametru metody, zdali id tokenu spadá do určeného rozsahu, viz. definice slovníku 2 a nahradí jej zobecněným tokenem, viz. 3, v tomto případě metoda zkontroluje, jestli je id tokenu v rozsahu mezi 3 až 9 kromě 7 (id pro `char`) a nahradí jeho typ typem `NUMBER` s id 200. Nahrazení probíhá v metodě `tokenization`.

Ostatní metody z rozhraní provádějí podobné nahrazení skupin symbolů, není proto nutné uvádět další podrobnosti. Ostatní algoritmy pro nahrazování posloupnosti tokenů vycházejí z nutnosti zmenšit počet porovnávaných tokenů na minimum, takže napří-

klad několikaúrovňové vkládání identifikátorů, ať už proměnných nebo metod přes jejich rodičovské třídy, je unifikováno do jednoho identifikátoru první metodou rozhraní `createUnifiedIdentifier`, prováděno nad celou kolekcí tokenů. Tato operace patří do druhé fáze tokenizace. Tu zastupuje operace `level2Tokenization`, proces, který zahrnuje ostatní operace z rozhraní.

Pro načtení jednotlivých tokenů do kolekce, která implementuje rozhraní `ArrayList` (které je mimochodem využíváno v aplikaci nejvíce), se v metodě `scanner`, jejíž vstupním parametrem je objekt reprezentující určitý typ souboru uloženého v dočasných adresářích, provádí několik kroků. Inicializace objektu `FileUnit` pro daný soubor, ošetření zachytávání výjimek, kde v bloku inicializujeme instanci třídy `Scanner` a přiřazujeme ji objekt stejného typu metodou `fileReading`, která provádí první lexikální analýzu znaků ze sady Unicode, v cyklu `while` poté načítáme jednotlivé tokeny, které přidáváme do kolekce objektu `FileUnit` metodou `add` a voláním `tokenization`, pro daný token, která provádí další část druhé fáze tokenizace. Po načtení všech tokenů a skončení cyklu se druhá fáze tokenizace dokončí a návratová hodnota tohoto procesu se ve formátu `FileUnit` předá dále. Tento proces je volán při inicializaci datové struktury `ProgramUnit`.

```
public FileUnit scanner(FileRep file) {
    FileUnit fileUnit = new FileUnit( file );
    Scanner scanner = fileReading(file.getFile() );
    while (!scanner.isZzAtEOF()) {
            Token token = scanner.scanToken();
            fileUnit .getTokens().add(tokenization(token));
            ...
  }
    level2Tokenization( fileUnit );
    return fileUnit ;
}
```

Výpis 6: Java skener naplňující FileUnit objekt se sekvencí tokenů

Zpracování importů

Z definovaného rozhraní `IJavaTokenization` (5) použijeme metodu oddělení importů `separateImports`, která nad daným objektem `FileUnit` vyextrahuje knihovny použité ve zdrojovém kódu z každého zpracovaného tokenu, který je typu `IMPORT`. Takto se přidávají importy do kolekce atributu `imports` z objektu `FileUnit`. Importy vygenerované z class souborů pomocí nástroje JAD nejsou relevantní pro srovnání, proto bude dektekce podobnosti na základě konfigurace programu probíhat na základě importů z java souborů.

Java bytekód skener

Lexikální analýza bytekódu probíhá na stejném principu jako tokenizace java kódu a implementuje rozhraní `IBytecodeTokenization` 7. V tomto případě používám tři slovníky pro práci s tokeny. V prvním slovníku jsou tokeny pro přečtení class souboru, druhý slovník obsahuje všechny instrukce bytekódu - opkódy, celkem 205 instrukcí, kde každá má svůj význam. V mezislovníku, za který označuji `IBytecodeDictionary`, jsou tedy všechny instrukce s tím rozdílem, že některé skupiny mají stejnou hodnotu. Díky reflexi tak mohu pracovat s jednou instrukcí z nějaké příbuzné skupiny a rovnou ji přiřadit konkrétní obecný token z třetího slovníku.

Skener čte na vstupu první token, instrukci `invokespecial`, kterou nahradí tokenem s ID = 12 a typem `IDENTIFIER` definovanou ve slovníku `IJavaBytecodeDictionary`, nyní se zavolá metoda `getBytecodeTokenID`, přijímající řetězec jako vstupní parametr, která hledá ve druhém slovníku `IBytecodeDictionary` podle řetězce vstupního parametru `invokespecial` příslušný název konstanty. Po přiřazení správného ID, které je ve všech případech instrukce začínající „invoke“ stejné (ID=52), vyhledá podle tohoto ID v posledním slovníku (`IGeneralizedBytecodeDictionary`) správnou obecnou skupinu označenou konstantou `INVOKE` s ID = 52. Takto se projde celá sekvence instrukcí a místo 205 instrukcí zůstane pro detekční proces pouze 60 skupin tokenů.

Problémem, který nastává při tokenizaci bytekódu je použití hybridního jazyka, který se skládá z části ze slovníku pro Javu a dále ze sady dvou set unikátních instrukcí se svými parametry. Objevují se další znaky a symboly, které se jinak nevyskytují ve struktuře zdrojového kódu Javy. Řešení je nastaveno na Javu verzi 1.7 class struktury souborů, která by měla být kompatibilní i s verzí jazyka 1.5.

```
public interface IBytecodeTokenization {
    ByteFileUnit byteScanner(FileRep file);
    int getBytecodeTokenID(String value);
    String getBytecodeTokenName(int token);
    String getMethodName(String value);
    Token instructionLineProcesing(List<Token> tokenLine);
    boolean isClass(List<Token> tokens);
    ...
    boolean isSwitch(List<Token> tokens);
    Token methodLineProcesing(List<Token> tokenBlock, int option);
    ByteFileUnit splitToMethods(List<Token> list, FileRep file );
    List<Token> tokensByLines(List<Token> tokens);
```

Výpis 7: Rozhraní IBytecodeTokenization pro implementaci metod ke skenování bytekódu

Implementované metody `fileReading` a `byteScanner` plní funkci stejnou jako u Java skeneru (provádějí skenování Unicode znaků a první fázi tokenizace metodou

scanToken). Hlavní myšlenkou druhé fáze tokenizace při zpracování bytekódu, který se skládá z částečné syntaxe Javy a instrukcí bytekódu, je nahrazování sekvencí tokenů po řádcích jejich nejobecnějším zástupným tokenem. Proces procházení řádků obstarává metoda tokensByLines, jejíž návratovou hodnotou je kolekce tokenů. Vně metody algoritmus rozpoznává jednotlivé sekvence tokenů a nahrazuje je jejich ekvivalentem ve slovníku (IGeneralizedBytecodeDictionary) podle jejich struktury. methodLineProcessing nahrazuje sekvence tokenů označujících metodu tokenem METHOD, pole jsou nahrazena konstantou FIELD, třída tokenem CLASS a statickou definici za STATIC. Jednotlivé instrukce metod jsou následně zpracovány další metodou instructionLineProcessing, kde klasická instrukce, skládající se z pozice instrukce, prefixu, který označuje datový typ, opcode instrukce a případného operandu, který odkazuje do constant poolu, je nahrazena příslušným obecným tokenem pro danou skupinu instrukcí. Další instrucke jsou rozhodovací větvení jako switch nebo tabulka výjimek. Ty jsou nahrazeny tokeny SWITCH nebo EXCEPTIONTABLE. Nakonec probíhá rozdělení me-

```
public ByteFileUnit byteScanner(FileRep file) {
  ...
        while (!scanner.isZzAtEOF()) {
            Token token = scanner.scanToken();
            fileUnit .getTokens().add(token);
        }
        List<Token> list = tokensByLines(fileUnit.getTokens());
  ...
        fileUnit  = splitToMethods( list ,  file ) ;
    return fileUnit ;
}
```

Výpis 8: Rozhraní IBytecodeTokenization pro implementaci metod ke skenování bytekódu

tod do kolekcí objektů BytecodeMethod, kterou jako objekt drží vyšší zástupce pro daný soubor formátu class - ByteFileUnit.

Proces skenování tedy transformuje textové řetězce ze souborů do sekvence tokenů, které jsou ve fázi detekce shody porovnávány. Spouštění tohoto procesu zahajuje inicializace objektu ProgramUnit, který spouští skenovací proces pro každý soubor java programu.

6.4 Vektor klíčových slov

Princip jak se vektor klíčových slov počítá a jeho význam je v kapitole 4. Jelikož se jedná o vektor, který uchovává všechny typy tokenů pro každou kategorii porovnávaných typů souborů, existuje jeden pro každou skupinu. Jeho inicializaci zahajuje aplikace vytvořením instance třídy ProgramGroup, která udržuje objekty typu ProgramUnit (pro každý porovnávaný vstupní program), z jejichž kolekcí FileUnit počítá jednotlivé vektory.

Aby se vypočítaly vektory klíčových slov pro srovnávané programy, musí se prvně inicializovat objekt `ProgramUnit`, který je nejobecnější datovou strukturou programu a uchovává informace o klíčových vektorech, importech z java souborů a samotné sekvence tokenů z obou srovnávaných programů, zadávaných při spuštění aplikace.

```
    private void fulfillKeywordVector () {
        Iterator it = getOriginal().getJavaList(). iterator ();
        FileUnit fileunit ;
        ByteFileUnit byteunit;
        while ( it .hasNext()) {
            fileunit = (FileUnit) it .next();
            this.javaKeyword.checkFileUnit(fileunit);
            this.javajadKeyword.checkFileUnit(fileunit);
        }
      ...
    this.javaKeyword.countFrequencies();
}
```

Výpis 9: Naplnění vektoru klíčových slov

Iterátor v kódu 9 prochází kolekce třídy `FileUnit`, kde se z každého objektu této kolekce nahrají všechny tokeny do vektoru. Vektor obsahuje všechny typy tokenů, které se vyskytují v tomto případě v souborech java, počítá jejich celkový počet pro každý typ a nakonec operací `countFrequencies` spočítá jejich frekvenci výskytu.

6.5 Matice podobnosti

Zdrojový kód pro výpočet matice podobnosti se kvůli jeho délce nachází v příloze 12. Vstupní parametry metody jsou vektor klíčových slov, parametry `alfa` a `beta`. Výsledkem výpočtu operace je matice s reálnými čísly, které odráží ohodnocení výskytu kombinace dvou typů tokenů v kódu.

```
private double[][] countSimilarityMatrix (KeywordVector vector, double alpha, double beta)
```

Tato metoda je volána z instance třídy `SimilarityMeasure`, která inicializuje a udržuje kolekci objektů třídy `Similarity`. Funkce pro výpočet matice podobnosti se vypočítává jednou pro každou instanci, takže v jednu chvíli existuje v aplikaci jeden vektor klíčových slov pro každou sadu porovnávaných souborů: java souborů, bytekód souborů, dekompilovaných souborů a případně pro srovnání souborů dekompilovaných a původních (java). Pro každou tuto skupinu tedy existuje jedna matice podobnosti, vypočítaná z daného vektoru. Hodnoty prvků matice jsou ovlivněny ladícími parametry `alfa` a `beta`.

6.6 Detekce shody

Zdrojový kód implementace algoritmu adaptivního lokálního zarovnání, který počítá podobnost dvou sekvencí tokenů (s využitím Smith-Watermanova algoritmu 3.4.2) lze nalézt v příloze 13. Tento algoritmus vypočítává adaptivní matici podobnosti, která vypočítává optimální lokální zarovnání. Samotné zarovnání ze sekvence tokenů provádí metoda Backtracing viz. 11. Celý proces je podrobně vysvětlen v kapitole 4.

Abychom provedli výpočet míry podobnosti pro dvě sady souborů z dané kategorie, musíme inicializovat třídu `SimilarityMeasure`, která v sobě udržuje kolekce objektů `Similarity`. Tato třída pak obsahuje objekt třídy `SimilarityMatrix` vizualizovavný jako UML třídní diagram, viz., 20.

Ta již obsahuje atributy a operace, které vypočítávají podobnost. Atribut `AMp_ab` je je celkový součet prvků adaptivní matice podobnosti, který udává absolutní hodnotu optimálního zarovnání mezi dvěma porovnávanými sekvencemi tokenů. `SM_ab` je matice podobnosti, `BT` je dvojrozměrná matice s informacemi o tom, z jakého předchozího prvku byl ten aktuální spočítán. Třída také obsahuje dvě kolekce zarovnaných sekvencí tokenů `alignment` a klíčový vektor `vector`.

Třídní operace `similarityDetection` volá metodu `SmithWatermanAlgorithm`, která počítá adaptivní lokální matici a poté vykoná algoritmus `backTracing`, jenž vypočte absolutní hodnotu podobnosti a vytvoří optimální lokální zarovnání mezi sekvencemi tokenů A a B.

Jelikož potřebujeme pro spočtení celkové míry podobnosti mezi dvěma sekvencemi tokenů (dvěma soubory) tři objekty třídy `SimilarityMatrix`, jak je vysvětleno v definici 4.4, jsou tyto objekty inicializovány v instanci třídy `Similarity` pro dvojice sekvencí $\{(A, B), (A, A), (B, B)\}$.

6.7 Výpočet míry podobnosti

Celkový výpočet podobnosti pro dva programy se inicializuje vytovřením instance třídy `ComplexProgramSimilarity`, která drží objekty třídy `SimilarityMeasure` pro každou skupinu porovnávaných souborů (mezi java soubory, bytekód soubory, dekompilovanými soubory a java s dekompilovanými). Tímto způsobem se postupně provede porovnávání podobnosti programů na všech úrovních.

Třída `SimilartyMeasure` obsahuje metodu `countSimilarities` (viz. 10), která prochází postupně vstupní kolekce souborových jednotek a postupně vytváří vektor objektů `Similarity`, kde se pomocí metody `similarityDetection` provede vyhodnocení adaptivní matice, zarovnání a výpočet míry podobnosti. Posledním procesem, který probíhá při zjišťování podobnosti je procentuální vyjádření podobností importů ze

```
private List<Similarity> countSimilarities(List<FileUnit> filesI, List<FileUnit> filesII,
    boolean align) {
    List<Similarity> matricies = new ArrayList<>();
    int length1 = filesI.size();
    int length2 = filesII.size();
    for(int i = 0;i<=length1-1;i++) {
        for(int j = 0;j<=length2-1;j++) {
            Similarity sm = new Similarity(this.alpha, this.beta, this.vector, filesI.get(i),
                filesII.get(j), this.SM, align);
            matricies.add(sm);
        }
    }
    return matricies;
}
```

Výpis 10: Iterace nad kolekcemi FileUnit objektů

skupiny java souborů zkoumaných programů. Pokud je zadána vysoká hodnota parametru `treshold`, vyloučí se ty instance podobností, které nemají podobnost stejnou nebo vyšší než je hodnota limitu.

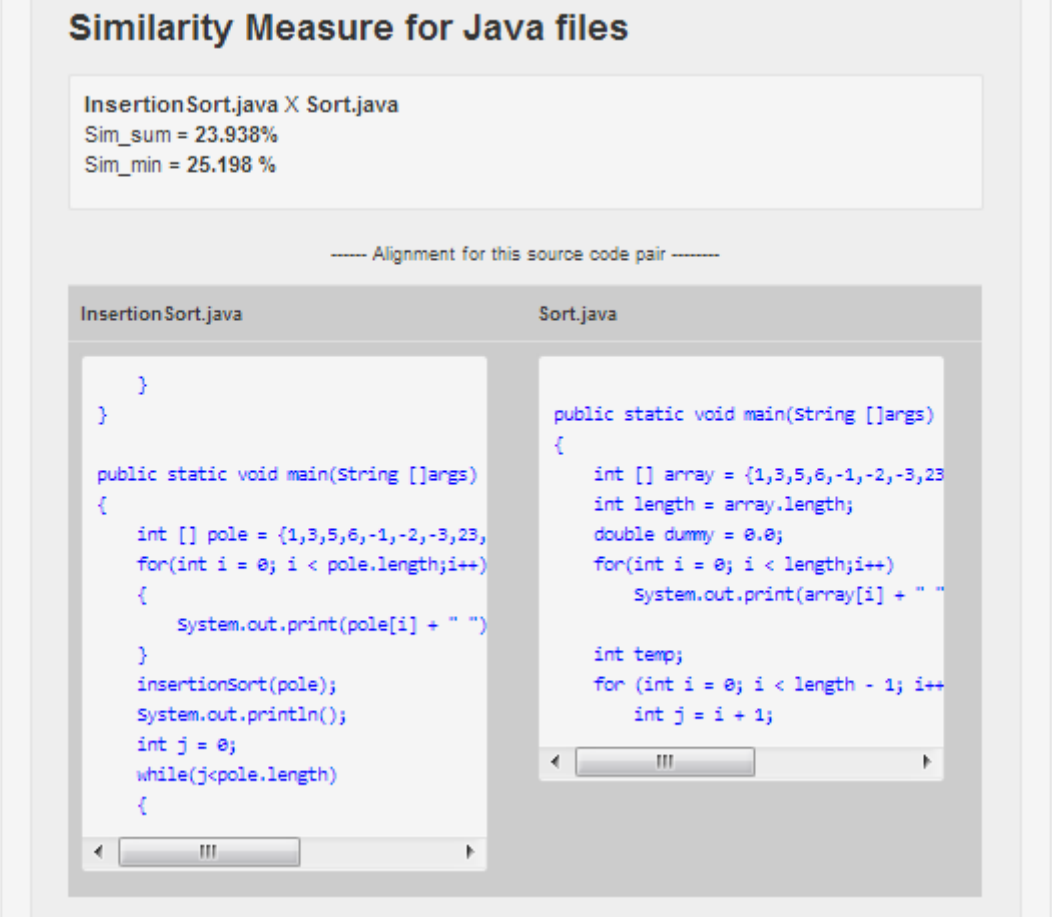
6.8 Mapování na původní zdrojové kódy

Abychom dokázali v poslední fázi vizualizovat podobné oblasti kódu, kde byla nalezena nejvyšší shoda, má každý objekt třídy `SimilarityMatrix` atributy `start1`, `start2`, `end1` a `end2`, které označují počáteční a koncové řádky z původních souborů. Tyto atributy jsou přístupné pomocí třídních „getterů“.

6.9 Vizualizace výsledků

Závěrečnou fází celého procesu detekce plagiátů v Java programech je vygenerování HTML reportu se statistikami zjištěných podobností. Třída `HTMLReport` pracuje s instancí `ComplexProgramSimilarity`, ze které má přístup ke všem výsledkům zjištěným při detekci. V tomto kroku využívám třídy pro čtení a zápis do souboru `FileReader` a `FileWriter` v kombinaci s třídami vlastností, které slouží pro výběrové čtení po řádcích `LineNumberReader` (pro nastavení, odkud číst kód z původního souborů) a bufferovaného zápisu po řádcích do HTML souboru.

Vizualizace popisuje zhodnocení pro danou úroveň (java, bytekód, atd.), udává procentuální podobnost SIM_sum a SIM_min pro porovnávané soubory a výpis fragmentu kódu s optimálním zarovnáním, viz., obr. 14. V případě úrovně bytekódu jsem nevytvářel výpis původního kódu namapovaného na zarovnání, protože se jedná o posloupnost instrukcí a její význam je hůře čitelný. Proto každá podobnost na úrovni metod bytekódu může být dohledána v java souborech. Podobnost importů byla vypočítána na základě postupu, viz, 4.4.



Obrázek 14: HTML výpis podobností aplikac PDA

6.10 Technologie a nástroje

Při implementaci jsem použil několik různých technologií. Nástroj pro dekompilaci bytekódu JAD [25], nástroj pro generování lexikálních analyzátorů, tzv. skenerů - JFlex [21], CUP (Parser Generator for Java) [26], jehož knihovny JFlex využívá a logovací knihovny pro Javu - Log4j [30] pro logování událostí v aplikaci (licence Apache Licence v2.0). Pro stylovou úpravu HTML reportu jsem použil framework Bootstrap [31], vydaný pod licencí Apache Licence v2.0.

Aplikace PDA byla vytvořena v integrovaném vývojovém prostředí NetBeans (verze 7.2.1) v programovacím jazyce Java (verze 1.7.0_21).

7 Testování aplikace

Testování aplikace vychází převážně z provedených testů několika skupin Java programů, jejichž podobnost byla vyhodnocena mezi systémy pro detekci plagiátů JPlag, který se používá na tomto poli poměrně často, navíc s dobrými výsledky, a mojí vlastní aplikací (PDA), která je postavena na podobném principu vnitřní reprezentace zdrojových kódů, ale na jiném algoritmu pro vyhodnocení míry podobnosti.

JPlag přináší také jiný typ výsledků ve vygenerované zprávě, protože vypočítává celkovou podobnost pro vstupní programy, PDA vyhodnocuje podobnosti na úrovni nižší, protože větší členitost porovnávaných jednotek zpřesňuje míru podobnosti mezi dvěma programy a jejich částmi. Testovací data jsou tedy rozděleny, podle skupin a srovnání je provedeno na úrovni souborů, aby výsledky JPlagu bylo možné porovnat s mým systémem.

7.1 Testovací data

Testovací data pro PDA byly shromážděny z několika zdrojů. První skupina programů byla uměle vytvořena několika studenty, kteří měli za úkol provést některé ze šesti úrovní modifikací a popsat tyto změny do zdrojového kódu a umožnit tak označení naleznutých pagiátů jako pravdivě pozitivních nebo jinak a zhodnotit tak úspěšnost systému. Tato skupina obsahuje programy: A0, A1, A2, A3. Abych mohl porovnat výsledky z obou detekčních systémů, musíme srovnávat na stejných úrovních, tedy jsou porovnávány jednotlivé třídy programů: A0, A1, A2, A3 a to: App, Mat a Reader.

Druhý zdroj dat pro testování jsou různé variace Smith-Watermanova algoritmu. Každá je implementovaná nezávisle a tudíž budeme moci posoudit, jak vypadá srovnání podobných zadání, a jak je vyhodnotí systémy pro detekci plagiátů. V této skupině se nachází programy: SW1, SW2, SW3 a SW4.

7.2 Metodologie testování

Testování JPlag systému probíhalo přes webového klienta, nahráním jednotlivých adresářů testovaných programů na server, jak vyžaduje aplikace. Bylo použito defaultní nastavení parametrů pro porovnávání. Nastavení parametrů vlastní aplikace je následující: `alpha` = 0,6, `beta` = 0,4 a `treshold` = 10 (vyřazuje hodnoty podobností menší než tento atribut). Podrobnosti o hodnotách parametrů ovlivňujících váhu shody/neshody dvou tokenů jsou popsány v 4.3.2. Hranice, kdy můžeme považovat dva programy za plagiarizované, je nad rozsahem mezi 30 - 40% [17]. Dále kategorizuji míru podobnosti do následujících rozsahů: Mírná: 35-60 %, Střední: 60 - 80 % a velká: 80 - 100 %, kdy 100 % znamená totožný zdrojový kód.

Testování má ukázat efektivitu a přesnost použitých procesů a algoritmů, popisovaných v této práci. Díky uměle vytvořených plagiarizovaných programů, podle zprávy v příloze A, můžeme ověřit přesnost vlastní aplikace a její úspěšnost při detekci plagiátů. Zde hledáme co nejvyšší počet pravdivě pozitivních nálezů, tedy nahlášené plagiáty jsou skutečně plagiáty a co nejmenší počet falešně negativních nálezů, což jsou dvojice, které jsou plagiarizované, ale unikly detekci. V případě druhé skupiny testovaných dat ověřujeme, zdali je systém odolný pro falešně pozitivním nálezům plagiátů.

Výsledky jsou zaneseny do tabulek podle programů A0 - A3 v rámci tříd, a programů SW1 - SW4. Tabulky obsahují procentuální hodnoty srovnání pro JPlag a aplikace PDA, která porovnává zdrojové kódy na úrovních: Java, dekompilovaných souborů (Jad) a Java s dekompilovanými (JavaJad). Podobnost metod v bytekódu je zaznamenána zvláště kvůli její členitosti porovnávání metod.

7.3 Srovnání PDA a JPlag

Podobnosti třídy App

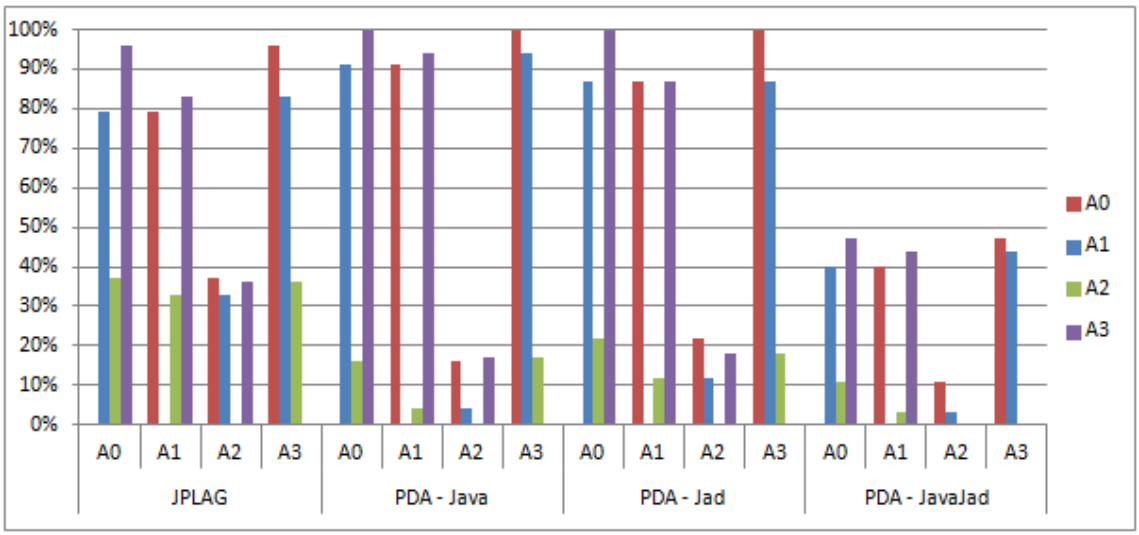
Podle výsledků z tabulky 4 zobrazených na grafu 15 vidíme, že třída App byla označena jako plagiarizovaná v programech A0, A1 a A3. U programu A2 byla zjištěna míra podobnosti na hranici plagiátu (nad 30 %) pouze u systému JPlag. PDA ani v jednom případě neoznačila třídu App programu A2 jako podezřelou. Ostatní programy byly označeny jako plagiáty s vysokou podobností na úrovních Java a Jad u PDA, JPlag vyhodnotil tyto programy se střední nebo vysokou mírou podobnosti, vždy o něco nižší než PDA. Jako totožné programy byly vyhodnoceny programy A3 a A0 na úrovni dekompilovaných a Java u systému PDA. Na úrovni JavaJad byly programy označeny jako mírně podobné.

	JPlag				PDA-Java				PDA-Jad				PDA-JavaJad			
	A0	A1	A2	A3	A0	A1	A2	A3	A0	A1	A2	A3	A0	A1	A2	A3
A0	x	79	37	96	x	91	16	100	x	87	22	100	x	40	11	47
A1	79	x	33	83	91	x	4	94	87	x	12	87	40	x	3	44
A2	37	33	x	36	16	4	x	17	22	12	x	18	11	3	x	0
A3	96	83	36	x	100	94	17	x	100	87	18	x	47	44	0	x

Tabulka 4: Procentuální podobnost třídy App

Pokud nahlédneme do zdrojových kódů programů A0 a A3 u třídy App zjistíme, že byly provedeny modifikace nejnižší úrovně, tj. změna formátování a přidání komentářů, které nemají vliv na vyhodnocení podobnosti. Jediný rozdíl byly importy, které JPlag bere v potaz pro srovnání a PDA je vyřazuje. Vyhodnocení na úrovni dekompilovaných souborů označilo programy jako totožné. Porovnání Java a dekompilovaných souborů (JavaJad) vykázalo pouze mírnou podobnost, po nahlédnutí do výpisu podobnosti fragmentů zdrojových kódu jsem zjistil, že dekompilovaný kód změnil začátek bloku

`try - catch` metodě main, tedy změnil pořadí několika příkazů a navíc se liší struktura metody systémového výpisu `println`, kde byla přidána třída `StringBuilder`. Tato změna zapříčinila snížení míry podobnosti.



Obrázek 15: Graf podobnostni třídy App mezi programy

Míra podobnosti na základě porovnání metod bytekódu z tabulky 5 potvrdily podobnosti programů A0, A1 a A3. Program A2 nebyl označen jako plagiovaný.

	Main - Main			
	A0	A1	A2	A3
A0	x	84	9	100
A1	84	x	0	81
A2	9	0	x	12
A3	100	81	12	x

Tabulka 5: Procentuální podobnost metod třídy App na úrovni bytekódu

Provedené modifikace úrovní L2 a L3, tedy přejmenování proměnných a přidání identifikátorů, nemělo příliš velký vliv na detekci plagiátů. Zato vytvoření nové metody, která implementovala část metody main a změna deklarace proměnných z lokálních na globální, v třídě App způsobila takovou míru modifikace, kterou systémy neodhalily.

Podobnosti třídy Mat

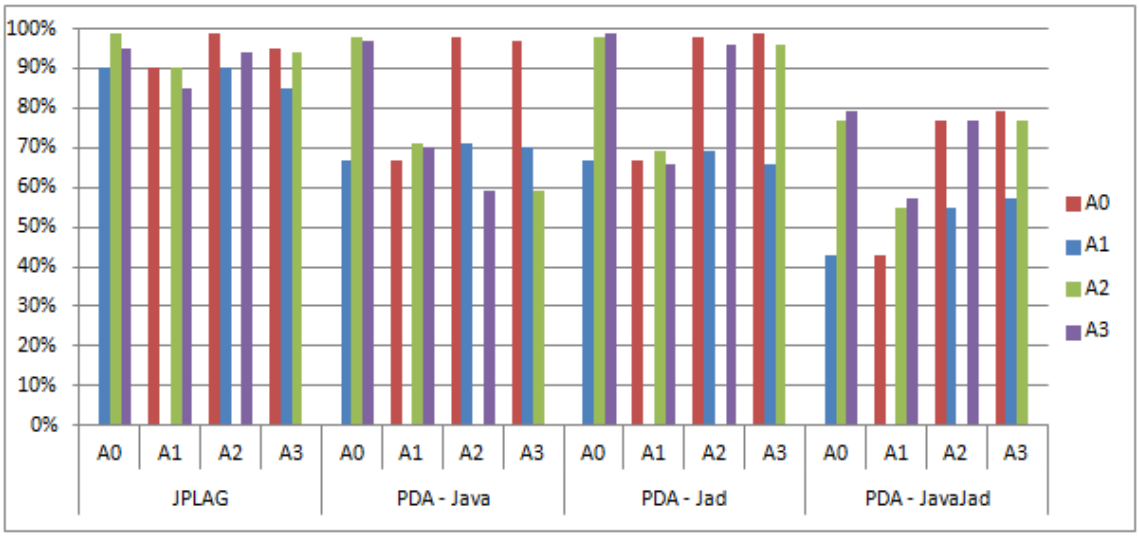
Na základě grafu 16, vytvořeného z tabulky 6, byly všechny programy A0 - A3 v rámci porovnání třídy Mat označeny jako plagiáty. JPlag označil všechny jako vysoce podobné

(nad 80%). PDA vyhodnotil programy A0, A2 a A3 jako vysoce podobné na úrovni dekompilovaných souborů, na úrovni Java byly označeny jako vysoce a středně podobné a na základě dekompilovaných a Java souborů byly vyhodnoceny jako mírně až středně podobné. PDA vyhodnotil program A1 nejlépe na 71 %. Po prozkoumání zdrojových kódů jsem zjistil, že modifikace byly provedeny od úrovně L1 až po L6, tedy nahrazení syntaktické struktury s ponecháním sémantiky a další změny, jako vkládání nedosažitelného kódu, apod. což zapříčinilo nižší výsledky detekce.

	JPlag				PDA-Java				PDA-Jad				PDA-JavaJad			
	A0	A1	A2	A3	A0	A1	A2	A3	A0	A1	A2	A3	A0	A1	A2	A3
A0	x	90	99	95	x	67	98	97	x	67	98	99	x	43	77	79
A1	90	x	90	85	67	x	71	70	67	x	69	66	43	x	55	57
A2	99	90	x	94	98	71	x	59	98	69	x	96	77	55	x	77
A3	95	85	94	x	97	70	59	x	99	66	96	x	79	57	77	x

Tabulka 6: Procentuální podobnost třídy Mat

Pokud se podíváme na hodnoty podobností na úrovni bytekódu v tabulce 7, zjistíme, že při vyšší členitosti úrovně porovnání zdrojových kódů získáme přesnější výsledky. Vezmeme-li v potaz program A1 v rámci třídy Mat, jsou výsledky podobnosti metod naznačující, že některé metody byly zkopírovány beze změny. Podobnosti dosahují 100 % u metod LoalL, LoadR, Square (u A1-A0 a A1-A3 porovnání) a u Gaussovy metody. V HTML reportu se nacházejí i další podobnosti, ale vybral jsem jen ty zajímavé.



Obrázek 16: Graf podobnosti třídy Mat mezi programy

Všimněme si hodnot 35 (65) u metody Singular. První hodnota určuje sumarizovanou míru podobnosti viz. 4.4 a v závorce hodnota 65 % vypočítaná podle 4.5. Tato `minimální` míra podobnosti předpokládá, že plagiarizovaná část kódu je vložena do kódu jiného, který může obsahovat další příkazy, nedosažitelný kód nebo jiné struktury a sumarizovaná podobnost by jej neoznačila jako plagiát. Takto je vyhodnocen se střední mírou podobnosti a při nahlédnutí do původního kódu vidíme, že kód byl modifikován na nejvyšší úrovni. Tedy porovnání na úrovni bytekódu může označit i těžko rozpoznatelný plagiát (JPlag označil tuto pasáž jenom z poloviny, ale v rozsahu celého souboru tato skutečnost neovlivnila odhalení plagiarizovaného kódu).

	Square			Singular			LoadL/LoadR			Gauss		
	A0	A2	A3	A0	A2	A3	A0	A2	A3	A0	A2	A3
A1	98	71	98	36 (65)	36 (65)	36 (65)	100	100	100	100	99	100

Tabulka 7: Podobnost metod na úrovni bytekódu třídy Mat

Pokud začleníme porovnávání metod na úrovni bytekódu do celkového souhrnu detekce plagiátů, PDA odhalil totožné pasáže zdrojového kódu u programu A1 s vysokou podobností (namísto středně podobných) a je tudíž velmi důležité, věnovat pozornost i výsledkům z této úrovně. Při manuální vyhodnocení takových zjištění je snadné si ověřit podobnosti v originálním souboru, kde uvidíme totožné fragmenty kódu, které byl plagiarizovány. I pokud neporovnáváme celý program nebo soubor, můžeme říci, že jeden byl zkopírován od druhého nebo naopak.

Podobnosti třídy Reader

Podle výsledků z tab. 8 a na základě grafu 17 byly zjištěny následující podobnosti. V rámci třídy Reader byly označeny, jako vysoce plagiované programy A0 s A2 u JPlagu a PDA - Jad. PDA - Java označil jako vysoce podobné tyto dvojice: A0 - A2 a A2 - A3. PDA - JavaJad označil A0 - A2 jako středně podobné. Na stejné úrovni podobnosti jsou označeny dvojice programů A0 - A1, A0 - A3, A1 - A2 a A2 - A3 u JPlagu. PDA - Java objevil podobnost střední u dvojice A0 - A3, mírně podobné u dvojic A1 - A2 a pod hranicí podobnosti A0 - A1 a A1 - A3. PDA - Jad označil předchozí dvojice vyšší hodnotou podobnosti, takže kromě A1 - A3 jsou ostatní označeny jako mírně až středně podobné. PDA - JavaJad označil jako možný plagiát pouze A0 - A2 se střední podobností a A0 - A1 a A0 - A3 jako mírně podobné. Ostatní byly označeny pod hranicí mírné podobnosti.

	JPlag				PDA-Java				PDA-Jad				PDA-JavaJad			
	A0	A1	A2	A3	A0	A1	A2	A3	A0	A1	A2	A3	A0	A1	A2	A3
A0	x	72	98	78	x	35	99	63	x	52	99	75	x	51	63	41
A1	72	x	72	41	35	x	48	39	52	x	51	37	51	x	38	23
A2	98	72	x	70	99	48	x	95	99	51	x	69	63	38	x	35
A3	78	41	70	x	63	39	95	x	75	37	69	x	41	23	35	x

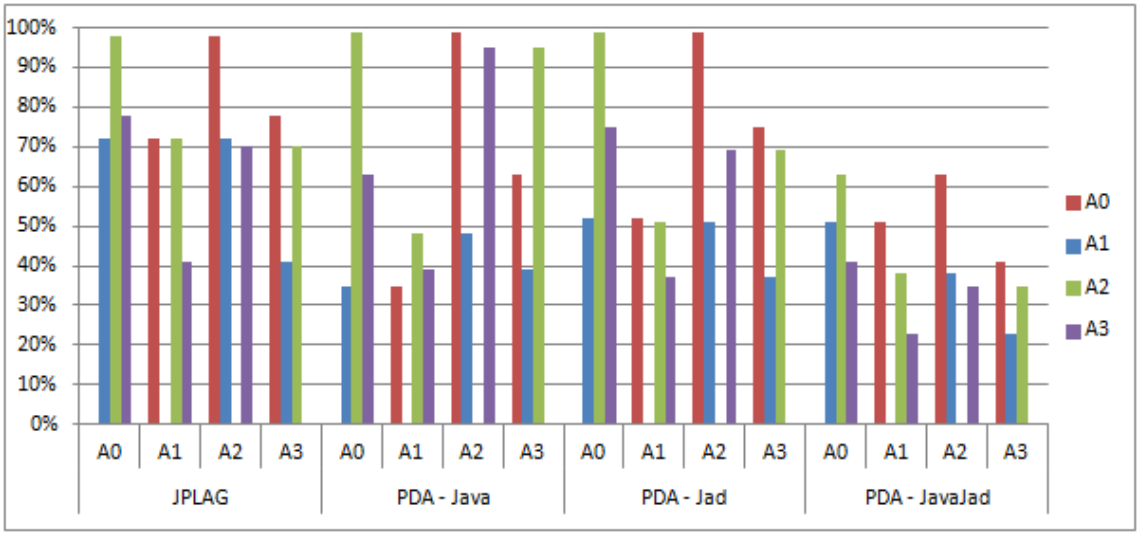
Tabulka 8: Procentuální podobnost třídy Reader

S nejnižší mírou podobnosti prošel testem program A1. Podíváme se tedy na jeho výsledky na úrovni bytekódu, viz. tab. 9. Podle výsledků můžeme tvrdit, že metoda readStr je plagiovaná mezi programy A1 - A0 a A1 - A2. Metodu open označil systém jako s mírnou podobností mezi A0 - A1, ale podle definice Sim_{sum} je tento kód podobný ze 72 %. Metoda length byla taktéž označena u programů A1 - A0 a A1 - A2 jako totožná. Mezi programy A1 - A3 byly nalezeny pouze nízké podobnosti. U jednotlivých tříd Reader

	readStr			open			length		
	A0	A2	A3	A0	A2	A3	A0	A2	A3
A1	85	83	43(63)	43(72)	19	20	100	100	48(60)

Tabulka 9: Podobnost metod na úrovni bytekódu třídy Reader

mezi programy byly provedeny tyto změny. Třída A1 obsahuje modifikace L3, L5 a L6, tedy úplné odstranění metody, převedení několika proměnných z lokálních na globální a o změnu logiky příkazu. U třídy A2 se jedná o modifikace nižších úrovní L1, L2, L3 a u třídy A3 se jedná o modifikace L2, L3 a L4 a L5 - globalizace proměnné, přidávání příkazů, odstranění bloku pro zachytávání výjimek.



Obrázek 17: Graf podobností třídy Reader mezi programy

Z testování vyšlo najevo, že s modifikacemi typu L1, L2 a L3 si poradily oba systémy a označily programy jako vysoce plagiované (A2). Srovnání PDA na úrovních Java a dekompilovaných souborů v některých případech odhalilo plagiáty s vyšší podobností než JPlag. Naopak při modifikacích, které zahrnovaly změnu pořadí nebo odstranění nějaké metody, byl úspěšnější JPlag než PDA systém. Abych zamezil zranitelnosti PDA vůči těmto druhům modifikací, navrhuji ve zhodnocení metodu ve fázi transformace, díky níž se zamezí přehlédnutí těchto plagiátorských technik.

Podobnosti importů

PDA má zabudovanou funkci pro výpočet podobností importovaných knihoven z java souborů. Tento výpočet je popisován v kap. 4.4. Z výsledků v tabulce 10 vidíme, že 78 % podobnost byla zaznamenána mezi programy A0 - A1 a A0 - A3, při nahlédnutí do zdrojových kódů pozorujeme rozdíly mezi několika vloženými třídami z balíčků tříd, ostatní jsou shodné. Podobnost A0 - A2 je 64 %, A2 - A1(A3) je rovna 54(53). Tato podobnost byla vyhodnocena v počtu okolo 10 vložených knihoven.

V rámci tohoto porovnání nejsou výsledky příliš směrodatné, ale pro testování postačí. Pokud zhodnotím předchozí výsledky podobností mezi programy A0 a A1, byly v předchozích částech výsledky méně uspokojivé v PDA - Java úrovni a zde vidíme skoro vysoce nalezenou podobnost. Může sloužit jako průvodce při podrobné analýze vývoje testovaných programů. Pokud bychom nalezli 90 až 100 % podobnost v malých programech, může být program označen jako podezřelý, v rámci detekce konfigurací, a postoupen k podrobnější analýze.

	A0	A1	A2	A3
A0	x	78	64	78
A1	78	x	54	64
A2	64	54	x	53
A3	78	64	53	x

Tabulka 10: Procentuální podobnost konfigurace (importovaných knihoven)

Návrhem pro zlepšení vyhledávání na úrovni konfigurací je oblast nastavení parametrů sestavení Java projektu v XML souborech, `build.xml`(skript `ant`) nebo `pom.xml`(Maven), což jsou nástroje pro správu softwarových projektů (sestavení projektu, vytváření reportů nebo dokumentace). Zde by mohl detekční nástroj vyextrahovat konkrétní parametry XMl souborů, pravidla pro sestavení, cíle apod. a porovnat jejich podobnost.

Podobnosti programů SW

Druhá skupina porovnávaných programů obsahuje 4 nezávislé implementace různých typů algoritmu Smith - Waterman. Programy se skládají z jednoho hlavního souboru s metodou main. Ve výsledcích tedy očekáváme malou nebo žádnou podobnost a ověřujeme, zdali systém nevyprodukuje falešně pozitivní nálezy, to jsou dvojice programů, které jsou nahlášeny jako plagiarizované, ale ve skutečnosti nejsou. Výsledky z tabulky 11 ukázaly, že systém JPlag neoznačil ani jeden z dvojic programů SW, jako skutečný plagiát. Nejvyšší míru podobnosti vykázala dvojice SW1 - SW4, a to 26 %. Tedy pod hranicí, kdy můžeme označit program za plagiarizovaný. PDA neodhalil ani na jedné úrovni žádnou podobnost. Procentuální vyjádření míry dosahovaly několika jednotek a výpis oblastní podobných fragmentů kódů byl zanedbatelný. Na úrovni SIM_{min} byly nalezeny

	JPlag			
	SW1	SW2	SW3	SW4
SW1	x	15	6	26
SW2	15	x	3	10
SW3	6	3	x	11
SW4	26	10	11	x

Tabulka 11: Procentuální podobnost SW programů

podobnosti v některých metodách, ale stále pod 40 %. Systémy neoznačily žádné dvojice jako plagiované, tedy počet falešně pozitivních nálezů je nulový a předpoklad byl splněn.

7.4 Srovnání výsledků

Na základě provedených testů jsem dospěl k následujícím závěrům. JPlag správně odhalil některé plagiáty s vysokou nebo střední mírou podobnosti, stejně jako tomu bylo u PDA na úrovni Javy. V první sadě programů byly správně označeny všechny plagiované programy kromě programu A2, který nebyl odhalen na žádné úrovni. U druhé sady nebyly detekovány plagiáty, jak bylo předpokládáno, jelikož programy vznikly nezávisle na sobě.

Srovnání úspěšnosti jednotlivých úrovní PDA, na kterých byla provedena detekce, ukázalo, že ve většině případů můžeme použít srovnání na úrovni Java, dekompilovaných souborů a bytekódu, kdy plagiáty odhalené JPlagem, byly taktéž označeny jako podezřelé na těchto úrovních. Někdy s větší mírou podobnosti, v případech použití modifikací, které neměnily ve velké míře pozice metod nebo příkazů, ostatní modifikace vyšší úrovně, ale v menším měřítku, neunikly odhalení, protože algoritmus lokálního zarovnání pokračoval ve výpočtu s postihem. V rámci bytekódu byla použitá metoda efektivnější, jelikož se podobnost vyhledávala mezi menšími jednotkami a tedy bylo odhaleno více plagiátů mezi páry programů.

Vizualizace nalezených shod systémem PDA zobrazuje optimální lokální zarovnání, vypočítané algoritmem a namapované na původní zdrojový kód. Vyznačené části zahrnují při nižších podobnostech i takový kód, který nemusel být zkopírován a upraven, anebo se jedná o vložený nedosažitelný kód.

Detekční systém PDA se nejhůře vypořádal s implementací nové metody modifikované z těla jiné metody. Jako řešení navrhuji několik rozšíření. Prvním je zmíněný popis statického trasování, popsaného v kap. 4.2. Tedy linearizace programu, která vede k odstranění těchto nedostatků, za cenu zvýšení výpočetní náročnosti.

Další možností je pozměnění algoritmu zpětného trasování tak, aby vytvořilo několik optimálních zarovnání v oblastech adaptivní matice podobnosti od nalezeného maxima

do konce vytvořené matice. Tyto oblasti by se daly rozdělit na tři potencionální regiony, kde by znovu proběhlo trasování od dalších maxim, pro dané lokální oblasti. Tak bychom získali zarovnání, která se nenalézala v původním výpočtu, jelikož byla překryta maximálním optimálním zarovnáním.

Poslední možností je rozčlenění porovnávaných jednotek programu na nižší struktury, jako metody. Tento přístup byl ověřen na úrovni bytekódu a je tedy východiskem i pro ostatní úrovně a možností, pro jejich vylepšení.

8 Závěr

Cílem práce bylo vypracování návrhu a implementace algoritmu, vyhodnocujícího podobnost softwarových artefaktů. Podobnost měla být vyhodnocena na několika úrovních: podobnost zdrojových kódu, podobnost konfigurace, např. použité knihovny, dále podobnost mezikódu a podobnost dekompilovaných kódů.

V úvodní, teoretické, části jsem se seznámil s problematikou plagiátorství a analyzoval detekční techniky a algoritmy, na jejichž základě jsem navrhoval algoritmus pro vyhodnocení podobnosti softwarových artefaktů.

V praktické části práce důkladně rozebírám detekční proces, který jsem použil v aplikaci, řešící zadanou problematiku. Zvolený detekční algoritmus, který vyhodnocuje podobnost na základě Smith - Watermanova lokálního zarovnání s ohledem na frekvenci výskytů klíčových slov v dané množině programů, jsem přizpůsobil pro vlastní detekční systém.

Implementovaná aplikace s integrovaným detekčním algoritmem vyhledává podobnosti mezi vstupními programy, které jsou rozloženy do požadovaných formátů a podobnost je vyhodnocena postupně na úrovni zdrojového kódu, bytekódu, dekompilovaných kódů a mezi konfigurací porovnávaných programů (importy knihoven a tříd).

V závěrečné části práce je popsáno testování implementované aplikace pro detekci plagiátů na uměle plagiovaných programech a ve skupině nezávisle vytvořených programů, implementujících stejný algoritmus. Výsledky jsou porovnány s detekčním nástrojem JPlag, aby byla objektivně ověřena úspěšnost použitých postupů a algoritmů. Zjištěné závěry z testování prokazují robustnost použití detekce podobnosti na různých úrovních, s výjimkou srovnání mezi úrovněmi původních zdrojových kódů a dekompilovaných kódů, kdy dekompilovaný kód vykazoval značné strukturální rozdíly, které snížily vyhodnocení míry podobnosti, oproti zdrojovému kódu.

Byly navrženy změny, které by odstranily tyto nedostatky. Aplikace může být využita při testování studentských projektů napsaných v Javě, rozšířena pro vyhodnocení podobností mezi dalšími programovacími jazyky nebo integrována do jiných systémů.

Ondřej Dočkal

9 Reference

[1] Jin-Su Lim, Jeong-Hoon Ji, Hwan-Gue Cho a Gyun Woo, *Plagiarism detection among source codes using adaptive local alignment of keywords*, Proceedings of the 5th International Conference [online]. 2011, č. 24, s. - [cit. 2013-01-30]. ISSN 978-1-4503-0571-6. DOI: 10.1145/1968613.1968643. Dostupné z: http://dl.acm.org/citation.cfm?doid=1968613.1968643

[2] Roy, Ch. K. a Cordy, J. R. *A Survey on Software Clone Detection Research* (2007) [cit. 2013-01-30]. School of Computing Queen's University at Kingston: Ontario, Canada. 2007:541.

[3] Roy, Chanchal K., James R. Cordy a Rainer Koschke. *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach*. Science of Computer Programming [online]. 2009, roč. 74, č. 7, s. 470-495 [cit. 2013-04-03]. ISSN 01676423. DOI: 10.1016/j.scico.2009.02.007. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/S0167642309000367

[4] Cosma, Georgina a Joy, Mike . *Source-code plagiarism: A UK academic perspectiv*, The University of Warwick, 2006 [cit. 2013-02-24]. DOI: 10.1.1.106.8344. Research Report No. 422. Department of Computer Science, The Univerity of Warwick.

[5] Černohlávková, Kateřina. *Plagiátorství na vysokých školách*. Brno, 2008 [cit. 2013-03-13]. Diplomová práce. Masarykova Univerzita.

[6] Jeong-Hoon Ji; Gyun Woo; Hwan-Gue Cho. *A Plagiarism Detection Technique for Java Program Using Bytecode Analysis*, Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on , vol.1, no., pp.1092,1098, 11-13 Nov. 2008 [cit. 2013-03-21]. DOI: 10.1109/ICCIT.2008.267. Dostupné z: http://dl.acm.org/citation.cfm?id=1471603.1471789

[7] Parker, A. a Hamblen, C. O. *Computer algorithms for plagiarism detection*. In: Education, IEEE Transactions on: vol.32, no.2, pp.94,99, [online]. 1989 [cit. 2013-03-21]. DOI: 10.1109/13.28038. Dostupné z: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=28038

[8] Hage, Jurriaan, Rademaker, Peter a Nike van Vugt. *A comparison of plagiarism detection tools*, Utrecht, 2010 [cit. 2013-03-30]. ISSN: 0924-3275. Dostupné z: www.cs.uu.nl. Technical Report UU-CS-2010-015. Utrecht University.

[9] *Novell lawsuit hits back at SCO*. In: Out-law.com [online]. 2005 [cit. 2013-03-22]. Dostupné z: http://www.out-law.com/page-5974

[10] Markoff, John. *Judge Says Unix Copyrights Rightfully Belong to Novell*. In: New York Times [online]. 2007 [cit. 2013-03-22]. Dostupné z: http://www.nytimes.com/2007/08/11/technology/11novell.html?_r=0

[11] Montalbano, Elizabeth. *Novell Won't Pursue Unix Copyrights*. In: http://www.pcworld.com [online]. 2007 [cit. 2013-03-22]. Dostupné z: http://www.pcworld.com/article/135959/article.html

[12] Shankland, Stephen. *Novell challenges SCO's Linux claims*. In: CNET News [online]. 2003 [cit. 2013-03-22]. Dostupné z: http://news.cnet.com/2100-1016-1010569.html

[13] *Plagiátorství - Infogram*. In: Infogram [online]. 2012 [cit. 2013-03-23]. Dostupné z: http://www.infogram.cz/

[14] Joy, Mike *Workshop on Plagiarism*. In: University of Bath, 2012 [cit. 2013-03-23].

[15] Clough, Paul. *Plagiarism in natural and programming languages*: an overview of current tools and technologies. Sheffield, 2000 [cit. 2013-03-24]

[16] Němečková, Lenka. *Plagiátorství*. ÚSTŘEDNÍ KNIHOVNA ČVUT. 2009 [cit. 2013-02-21], 7 s. Dostupné z: http://knihovna.cvut.cz/studium/jak-psat-vskp/doporuceni/plagiatorstvi/

[17] Hic, Daniel *Detekce plagiovaného software*. Ostrava, 2012 [cit. 2013-04-01]. Diplomová práce. VŠB - Technická univerzita Ostrava, Fakulta elektrotechniky a informatiky.

[18] Brixtel, R., Robbes, R., Bazin, C., Lesner, B. a Fontaine, M. *Language-Independent Clone Detection Applied to Plagiarism Detection*. In: Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on [online]. 2010 [cit. 2013-03-27]. ISBN 978-1-4244-8655-7. DOI: 10.1109/SCAM.2010.19. Dostupné z: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5601829

[19] Gasevic, Dragan a Djuric, Zoran. *A Source Code Similarity System for Plagiarism Detection*. Athabasca, Canada, 2012 [cit. 2013-03-27], 42 s. Dostupné z: https://files.semtech.athabascau.ca/public/TRs/TR-SemTech-07022012.pdf

[20] *Tokens and Java Programs*. Northern Michigan University [online]. 2000 [cit. 2013-04-03]. Dostupné z: http://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/tokens/lecture.html

[21] Klein, Gerwin *JFLEX: The Fast Scanner Generator for Java* [online]. 2009 [cit. 2013-04-03]. Dostupné z: www.jflex.de

[22] Kapser, Cory J *Toward an Understanding of Software Code Cloning as a Development Practice*. Waterloo, Canada, 2009 [cit. 2013-04-11]. Disertační práce. Univerzita Waterloo.

[23] Wise, Michael J. *String Similarity via Greedy String Tiling and running Karp-Rabin Matching* [online]. Sydney, Australia, 1993 [cit. 2013-04-06]. Dostupné z: http://opstools.googlecode.com/svn-history/r31/trunk/ndd/paper/. Článek. Univerzita v Sydney.

[24] Bortolussi, Luca *Suffix Tree: From exact to approximate string matching*. 2003 [cit. 2013-04-10]. Dostupné z: www.dmi.units.it/~bortolu/files/slides/

[25] Kouznetsov, Pavel *Jad: the fast Java Decompiler*. 2001 [cit. 2013-04-30]. Dostupné z: http://www.varaneckas.com/jad/

[26] Hudson, Scot, Frank Flannery, C. Scott Ananian *CUP: A Parser Generator for Java*. Princeton, 1999 [cit. 2013-04-30]. Dostupné z: http://www.cs.princeton.edu/~appel/modern/java/CUP/

[27] *Defenice plagiátu*. In: Infogram [online]. 2013 [cit. 2013-03-25]. Dostupné z: http://www.infogram.cz/findInSection.do?sectionId=1115&categoryId=1161

[28] TDKIV - *Česká terminologická databáze z oblasti knihovnictví a informační vědy* [online]. [cit. 2013-03-22]. Dostupné z: http://aleph.nkp.cz/cze/ktd.

[29] Mareš, Jiří. *Elektronické podvádění ve škole*. 2005 [cit. 2013-03-30]. Dostupné z: http://www.latal.cz/management/userfiles/elektronicke_podvadeni_ve_skole-mares_jiri.pdf

[30] The Apache Software Foundation *Apache log4j™ 1.2: Apache logging services*. 2013 [cit. 2013-04-30]. Dostupné z: http://logging.apache.org/log4j/1.2/.

[31] Otto Mark, Thornton, Jacob *Bootstrap*. Sleek, intuitive, and powerful front-end framework for faster and easier web development. 2012 [cit. 2013-04-30]. (Version 2.3.1) [Software]. Dostupný z: http://twitter.github.io/bootstrap/index.html.

A Příloha: Zpráva o úpravě Java kódů pro testování detekce plagiátů

Tato zpráva popisuje možnosti modifikace zdrojových kódů napsaných v programovacím jazyce Java. Cílem je vytvoření věrohodných umělých plagiovaných dat pro testovaní aplikace pro detekci plagiátů. Náhledem výstupu (tedy plagiarizovaných kódů) je obr. 18, zachycující provádění změn v přiloženém testovacím projektu. Zaznamenání úprav je důležité pro ohodnocení úspěšnosti detekční aplikace (recall a precision).

Úroveň modifikací

Úrovně modifikací, které lze použít při umělém plagiování, podle kategorií.

- L0: Úroveň 0 znamená zkopírování kódu bez jakýchkoliv úprav. Nejjednodušeji odhalitelný.
- L1: Úrovně 1 zahrnuje úpravu komentářů a formátování kódu. Jedná se o vizuální změny, jako úprava prázdných řádků, tabulátorů a mezer. Komentáře jsou parafrázovány, přidány, přeloženy nebo naopak úplně odstraněny.
- L2: Úroveň 2 je změna názvů identifikátorů. Jedná se o přejmenování proměnných nebo metod.
- L3: Modifikace na úrovni 3 zahrnuje úpravu deklarace proměnných. Může se jednat o změnu datového typu, umístění deklarace proměnné, přehození pořadí operandů v příkazu bez ovlivnění výsledku nebo záměna globálních a lokálních proměnných.
- L4: 4 úroveň se týká změny pořadí metod nebo funkcí. Plagiátor může pozměnit zdrojový kód nahrazením volání metody implementací jejího těla a naopak, sloučit více metod do jedné, to samé platí o třídách a souborech, které program používá.
- L5: Úroveň 5 se týká významnějších zásahů do původního kódu, ovlivňují vizuální a syntaktickou strukturu kódu. Jedná se o vkládání nadbytečných příkazů a metod, které v zásadě nemění funkcionalitu kódu, přepisování syntaktických struktur selekce a iterace nebo vkládání nedosažitelného kódu.
- L6: Poslední úroveň mění kompletně syntaktickou strukturu některých částí nebo celého programu. Sémantika je zachována a tedy i konzistence původní logiky programu. Pod takovoou záměnou si můžeme představit místo použití rekurzivní funkce faktoriál, posloupnost příkazů s cyklem for.

Dokumentace úprav kvůli ověření správnosti detekovaných podobností: V částech, kde modifikujete kód, uveďte v komentáři typ úpravy. viz. obr. 18.

IDE

Testovací projekt byl vytvořen v NetBeans IDE 7.2 a proto také doporučuji použít toto integrované vývojové prostředí. Případně lze zpracovat v Eclipse nebo jiném IDE.

```java
public class App {

    static int faktorial(int n) {
        int a = 1;
        for (int i = 1; i <= n; i++)
            a = a * i;
        return a;
    }

    static double deleni(int a, int b) {
        if (b == 0)
            System.out.println("Nelze delit nulou");
        else
            return (double)a / (double)b;
        return 0;
    }

    static void vypis(int n) {
        if (n < 10)
            System.out.println("Mensi: " + n);
        else
            System.out.println("Vetsi: " + n);
    }

    public static void main(String[] args) {
        int number;
        int VALUE = number = 5;
        int a;
        int delitel = 7;
        double d;

        a = faktorial(VALUE);
        vypis(a);

        d = deleni(a,delitel);
        System.out.println("Vysledek deleni: " + d);

        for(int x = 1;x<10;x++)
        {
            number = number * x;
        }
    }
}
```

```java
public class Application {
    //L3 misto tridni promenne deklarujeme globalni
    static final int DELITEL = 7;
    /*
     * L3 - zmena nazvu metody oproti puvodni, zmena podminky
     * L4 - zmena poradi umisteni metody
     * L1 - zmena formatovani kodu + pridany komentare
     */
    public static void print(int n) {
        if (n > 10)
        {
            System.out.println("Vetsi nez 10: " + n);
        }
        else
        {
            System.out.println("Mensi nez 10: " + n);
        }
    }
    /*
     * L6 - syntakticka zmena struktury tela metody,
     * vyuziti rekurze oproti
     * iteraci, semantika je stejna
     */
    static int faktorial(int n) {
        if (n == 0)
            return 1;
        else return n * faktorial(n - 1);
    }
    public static void main(String[] args) {
        //L2 a L3 - umisteni deklaraci, zmena nazvu promennych,
        //datovych typu
        int a,n,VALUE = 5;
        n = VALUE; float d;
        a = faktorial(VALUE);
        print(a);
        //L4 - misto volani metody vlozeni jejiho tela
        if((DELITEL)!=0) {d = (float)a/(float)DELITEL;}
        else {
            System.out.println("Nelze delit nulou");
            d = 0;}
        System.out.println("Vysledek deleni: " + d);
        //L5 - zmena kontrolni struktury cyklu
        int x = 1;
        while(x<10) {
            n = n * x;
            x++;}
    }
}
```

Obrázek 18: Ukázka modifikací v Javě

Závěr

Zpráva popisuje možnosti úprav zdrojových kódů podle úrovní a dále uvádí postup zaznamenávání provedených modifikací v kódu jazyku Java pro testování plagiátů.

B Příloha - zdrojové kódy

V příloze se zdrojovými kódy se nalézají tři stěžejní algoritmy pro výpočet adaptivního lokálního zarovnání. Výpočet matice podobnosti 12, Smith - Watermanův algoritmus pro matici adaptivního lokálního zarovnání 13 a algoritmus pro zpětné trasování optimálního zarovnání backtracing 11.

```
    private void backTracing(double[][] M, List<Token> f1, List<Token> f2, int[][] BT) {
        double max = 0.0;
        int line = 0, column = 0;
        int i, j;
        // tracing back the pathway;
        for (i = 0; i <= f1.size(); i++) {
            for (j = 0; j <= f2.size(); j++) {
                if (M[i][j] > max) {
                    max = M[i][j];
                    line = i;
                    column = j;
                }
            }
        }
        i = line;
        j = column;

        if (this.isAlignment) {
            while ((i > 0) && (j > 0)) {
      //diagonal alignment
                if (BT[i][j] == 0) {
                    this.getAlignment_1().add(f1.get(i - 1));
                    this.getAlignment_2().add(f2.get(j - 1));
                    this.AMp_ab += M[i][j];
                    i = i - 1;
                    j = j - 1;
                    //deletion
                } else if (BT[i][j] == 1) {
                        this.getAlignment_1().add(f1.get(i - 1));
                        this.getAlignment_2().add(new Token(f2.get(j).getLine(), -1, "GAP", "--",
                                ""));
                        this.AMp_ab += M[i][j];
                    i = i - 1;
                    // insertion
                } else if (BT[i][j] == -1) {
                        this.getAlignment_1().add(new Token(f1.get(i).getLine(), -1, "GAP", "--",
                                ""));
                        this.getAlignment_2().add(f2.get(j - 1));
                        this.AMp_ab += M[i][j];
                    j = j - 1;
                }
            }
    }
```

Výpis 11: Algoritmus backtracing procesu pro tuto metodu

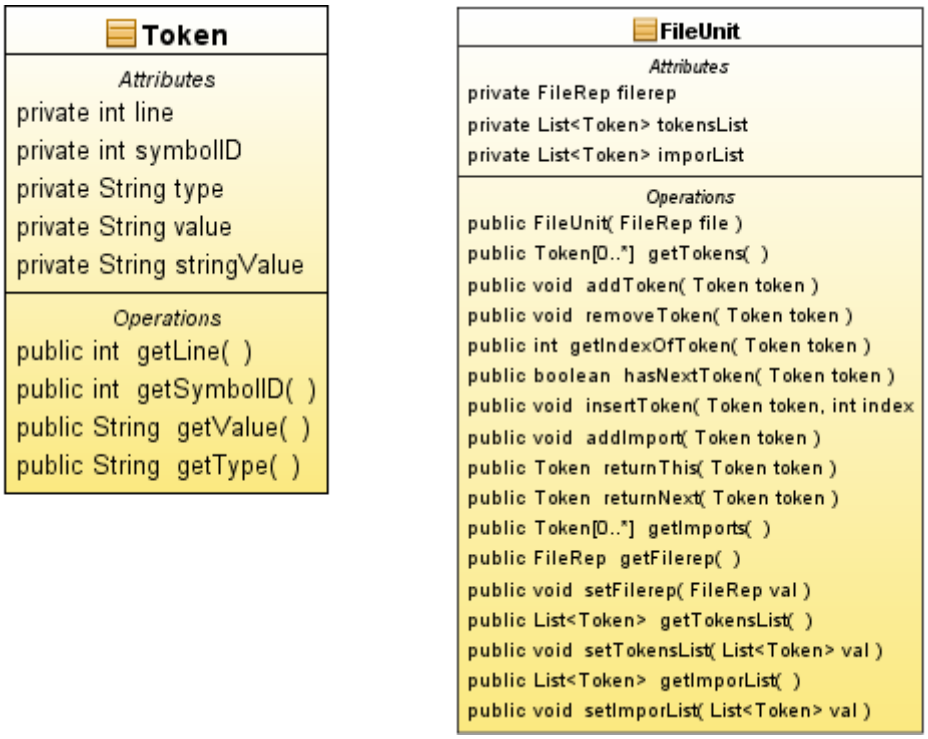
```
private double[][] countSimilarityMatrix(KeywordVector vector, double alpha, double beta) {
    int size = vector.getVector().size();
    double fi, fj, logs;
    int ii, jj;
    /* initializating the similarity matrix of size r+1, where r is the number
     * of kinds of keywords and +1 is the column and row for gap symbol */
    double[][] M = new double[size + 1][size + 1];
    /*Negative infinite value for the left-most upper element which represent
    * situation where both keywords are gaps */
    M[size][size] = Double.NEGATIVE_INFINITY;
     // Fulfilling the last row with values where kj is a gap
    for (int i = 0; i < size; i++) {
        //Deletion, kj is a gap, ki is not
        //4.Beta.log2(fi)
        fi = vector.getFrequencyOfKeywordByIndex(i);
        M[i][size] = 4 * beta * (Math.log(fi) / Math.log(2));
    }
    // Fulfilling the last column with values where ki is a gap
    for (int j = 0; j < size; j++) {
        //Insertion, ki is a gap, kj is not
        //4.Beta.log2(fi)
        fj = vector.getFrequencyOfKeywordByIndex(j);
        M[size][j] = 4 * beta * (Math.log(fj) / Math.log(2));
        //System.out.println("M[" + size + ","+ j + "] = " + M[size][j]);
    }
     /* Fulfilling the rest of the similarity matrix
     * Adding values for matching and mismatching pairs of keyword */
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            ii = vector.getSymbolIDByIndex(i);
            jj = vector.getSymbolIDByIndex(j);
            fi = vector.getFrequencyOfKeywordByIndex(i);
            fj = vector.getFrequencyOfKeywordByIndex(j);
            logs = (Math.log(fi * fj) / Math.log(2));
            //Match
            if (ii == jj) {
                M[i][j] = (-1) * alpha * logs;
            }
  //Mismatch
            else if (ii != jj) {
                M[i][j] = beta * logs;
            }
        }
    }
    return M;
}
```

Výpis 12: Metoda pro výpočet matice podobnosti

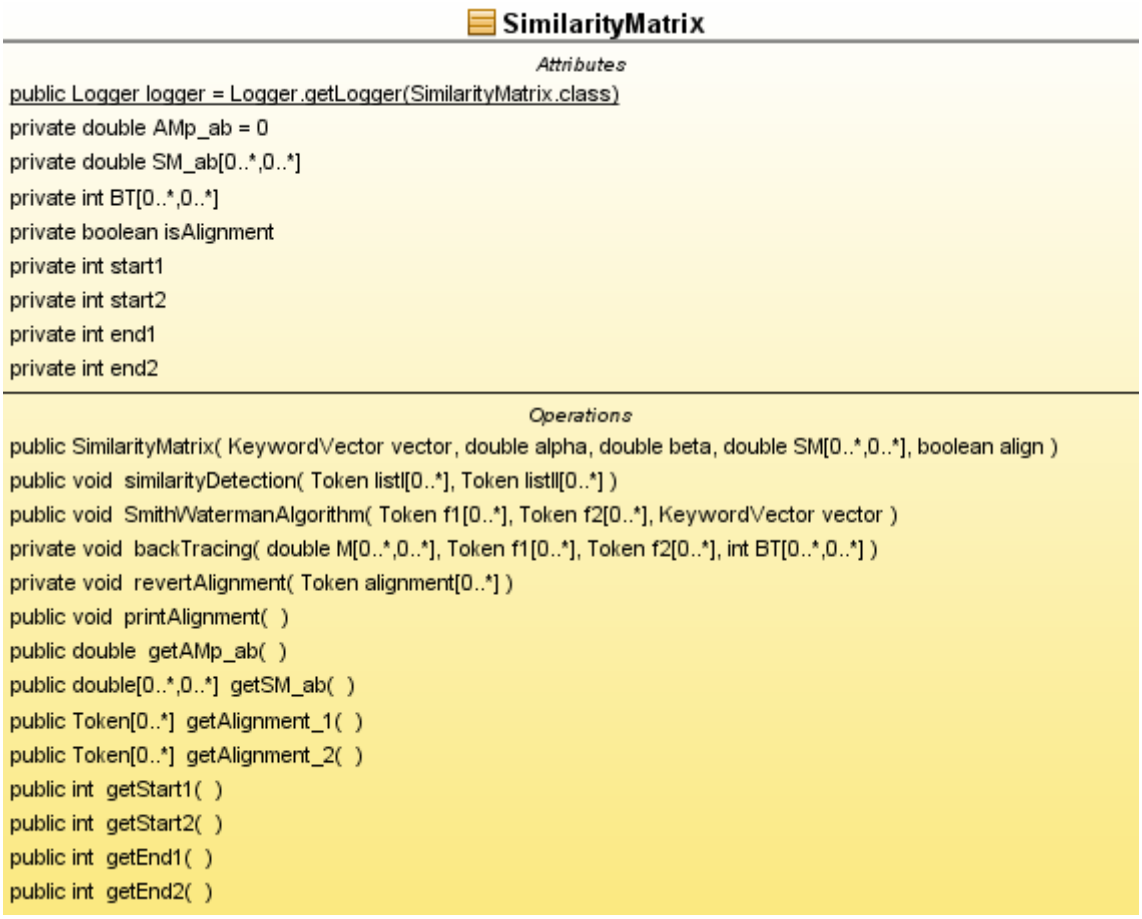
```
public void SmithWatermanAlgorithm(List<Token> f1, List<Token> f2, KeywordVector vector)
    {
    double m;
    //counting similarity matrix with keyword vector given and alpha and beta
    //index of the last element of the similarity matrix
    int last = vector.getVector().size() - 1;
    //match, mismatch, delete, insert;
    double match, mismatch, delete, insert;

    //defining the H matrix for local alignment, with extra column and row for gaps
    double[][] H = new double[f1.size() + 1][f2.size() + 1];
    //tracing matrix for recording the choosen path and consecutive backtracking
    // if diagonal value is maximum - 0 is added - match/ mismatch
    // if upper value is maximal, 1 is added - deletion
    // if left value is maximal, -1 is added - insertion
    this.BT = new int[f1.size() + 1][f2.size() + 1];

    //adding zeros to 0-th row and column
    for (int j = 0; j <= f2.size(); j++) {
        H[0][j] = 0;
        this.BT[0][j] = 2;
    }
    for (int i = 0; i <= f1.size(); i++) {
        H[i][0] = 0;
        this.BT[i][0] = 2;
    }

    for (int i = 1; i < f1.size(); i++) { // assigned the optimal score for the alignment;
        for (int j = 1; j < f2.size(); j++) {
            //SM is the score of the matched keyword from similarity matrix
            //indexes of the keyword in vector
            int idkey = vector.getIndexOfToken(f1.get(i).getSymbolID());
            int jdkey = vector.getIndexOfToken(f2.get(j).getSymbolID());
            //assigning values to its possibilities of alignment
            match = H[i - 1][j - 1] + this.SM_ab[idkey][idkey];
            mismatch = H[i - 1][j - 1] + this.SM_ab[idkey][jdkey];
            delete = H[i - 1][j] + this.SM_ab[idkey][last];
            insert = H[i][j - 1] + this.SM_ab[last][jdkey];
            //checking whether compared tokens are the same
            if (f1.get(i).getSymbolID() == f2.get(j).getSymbolID()) {
                m = match;
            } else {
                m = mismatch;
            }
            if ((m > delete) && (m > insert) /*&& (m > 0)*/) {
                H[i][j] = m;
                BT[i][j] = 0;
            } else if ((delete > insert) /*&& (delete > 0)*/) {
                H[i][j] = delete;
                BT[i][j] = 1;
            } else {
                H[i][j] = insert;
                BT[i][j] = -1;
            }
        }
    }

    backTracing(H, f1, f2, BT);
}
```

Výpis 13: Adaptivní lokální zarovnání s použitím SW algoritmu

C Přílohy - Obrázky



Obrázek 19: UML třídní diagram tříd FileUnit a Token



Obrázek 20: UML třídní diagram třídy SimilarityMatrix

D Přílohy - Přiložené médium

D.1 Adresářová struktura CD

- `PD-Application.zip` - komprimovaný soubor se spustitelnou `jar` aplikací
- `\PD-Application` - adresář s Java aplikací, kterou je možné otevřít v IDE
- `\Others` - adresář se specifikačními soubory pro JFlex generátor skenerů
- `\TestProjects` - adresář s testovacími data, obsahuje skupiny GroupA, GroupB a dva testovací projekty

D.2 Požadavky na spuštění aplikace

Aplikace PDA byla vytvořena v integrovaném vývojovém prostředí NetBeans (verze 7.2.1) v programovacím jazyce Java (verze 1.7.0_21). Požadavky na spuštění aplikace jsou:

- Java SE Runtime enviroment (build 1.7.0_21-b11)
- Připojení systémové cesty na adresář `bin` s příkazy JDK (aplikace spouští příkaz `javap`).
- Stáhnutý nástroj JAD z [25] a navedení jeho adresáře do systémové cesty (proměnná prostředí `PATH`)
- Spuštění `jar` souboru z přílohy (PD-Application.jar) se provádí příkazem „java -jar PD-Application.jar [Cesta1] [Cesta2]“, pokud jsme aktuálním adresáři `dist` aplikace, a kde parametry jsou cesty k testovaným programům, nebo můžeme aplikace spustit bez parametrů, ale musí se upravit soubor `config.properties`, kde se zapíšou relativní nebo absolutní cesty k programům a nastaví se parametry.
- Aplikace přistupuje do lokální složky a zapisuje do souborů, je tedy nezbytné, aby nebyla spouštěna z média
- Komprimovaná aplikace se nachází v adresáři `dist`, a můžeme ji spustit výše zmíněnými kroky. Nebo otevřít projekt PD-Application v IDE jako existující projekt.